

# TOTALVIEW

## USERS GUIDE



AUGUST 2001  
VERSION 5.0

Copyright © 1999–2001 by Etnus LLC. All rights reserved.

Copyright © 1998–1999 by Etnus Inc. All rights reserved.

Copyright © 1996–1998 by Dolphin Interconnect Solutions, Inc.

Copyright © 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of Etnus LLC (Etnus).

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Etnus has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Etnus. Etnus assumes no responsibility for any errors that appear in this document.

TotalView and Etnus are registered trademarks of Etnus LLC. TimeScan and Gist are trademarks of Etnus LLC.

All other brand names are the trademarks of their respective holders.



# Contents

## About This Book

Supported Platforms .....	xv
Reporting Problems .....	xvi
Conventions .....	xvi

## 1 TotalView Features

TotalView Advantages .....	1
TotalView Windows .....	3
Understanding Multiprocess Programs .....	5
Understanding Multithreaded Programs .....	6
Controlling Processes and Threads .....	7
Using Action Points .....	8
Examining and Manipulating Data .....	9
Distributed Debugging .....	10
Visualization .....	11
Context-Sensitive Help and Documentation .....	13

## 2 TotalView Basics

Compiling Programs .....	15
Starting TotalView .....	16
Initializing the Debugger .....	17
Using the Mouse Buttons .....	18
Using the Root Window .....	19
The Process Window .....	22
Starting a Process .....	22
Diving into Objects .....	26
Editing Text .....	28
Searching for Text .....	28
Searching for Functions and Variables .....	29

Saving the Contents of Windows .....	30
Exiting from TotalView .....	31
<b>3 Setting Up a Debugging Session</b>	
Compiling Programs .....	33
Starting TotalView .....	34
Loading Executables .....	36
More on Loading Remote Executables .....	37
Reloading a Recompiled Executable .....	38
Attaching to Processes .....	39
Attaching Using the Unattached Page .....	39
Attaching Using File > New Program .....	40
Detaching from Processes .....	41
Examining a Core File .....	42
Processes and Thread State .....	43
Attached Process States .....	44
Unattached Process States .....	45
Handling Signals .....	45
Setting Search Paths .....	48
Setting Command Arguments .....	50
Setting Input and Output Files .....	51
Setting Preferences .....	52
Setting Preferences, Options, and X Resources .....	54
Setting Environment Variables .....	58
Monitoring TotalView Sessions .....	59
<b>4 Setting Up Remote Debugging Sessions</b>	
Starting the TotalView Debugger Server .....	61
Single Process Server Launch Options .....	62
Bulk Launch Window Options .....	63
Starting the Debugger Server Manually .....	65
Single Process Server Launch Command .....	66
Bulk Server Launch on an SGI MIPS Machine .....	68
Bulk Server Launch on an IBM RS/6000 AIX Machine .....	69
Disabling Autolaunch .....	70
Changing the Remote Shell Command .....	70
Changing the Arguments .....	71
Autolaunch Sequence .....	71
Debugging Over a Serial Line .....	72
Start the TotalView Debugger Server .....	73

Starting TotalView on a Serial Line .....	74
New Program Window .....	74

## 5 Setting Up Parallel Debugging Sessions

Debugging MPI Applications Overview .....	77
Debugging MPICH Applications .....	78
Starting TotalView on an MPICH Job .....	79
Attaching to an MPICH Job .....	80
MPICH P4 procgroup Files .....	81
Debugging Compaq MPI Applications .....	82
Starting TotalView on a Compaq MPI Job .....	82
Attaching to a Compaq MPI Job .....	83
Debugging HP MPI Applications .....	83
Starting TotalView on an HP MPI Job .....	83
Attaching to an HP MPI Job .....	84
Debugging IBM MPI (PE) Applications .....	84
Preparing to Debug a PE Application.....	84
Switch-Based Communication .....	85
Remote Login .....	85
Timeout .....	85
Starting TotalView on a PE Job .....	86
Setting Breakpoints .....	86
Starting Parallel Tasks .....	86
Attaching to a PE Job .....	87
Attaching from a Node Running poe .....	87
Attaching from a Node Not Running poe .....	88
Debugging QSW RMS Applications .....	88
Starting TotalView on an RMS Job .....	88
Attaching to an RMS Job.....	89
Debugging SGI MPI Applications .....	89
Starting TotalView on a SGI MPI Job .....	89
Attaching to an SGI MPI Job .....	90
Displaying the Message Queue Graph .....	90
Displaying the Message Queue .....	92
Message Queue Display Overview .....	92
Message Operations .....	93
MPI Process Diving .....	93
MPI Buffer Diving .....	94
Pending Receive Operations .....	94
Unexpected Messages .....	94

Pending Send Operations .....	95
MPI Debugging Troubleshooting .....	95
Debugging OpenMP Applications .....	96
Debugging an OpenMP Program .....	97
OpenMP Private and Shared Variables .....	98
OpenMP THREADPRIVATE Common Blocks .....	101
OpenMP Stack Parent Token Line .....	102
Debugging PVM and DPVM Applications .....	103
Supporting Multiple Sessions .....	103
Setting Up ORNL PVM Debugging .....	104
Starting an ORNL PVM Session .....	104
Starting a DPVM Session .....	106
Automatically Acquiring PVM/DPVM Processes .....	107
Attaching to PVM/DPVM Tasks .....	108
Reserved Message Tags .....	109
Debugging Dynamic Libraries .....	110
Cleanup of Processes.....	110
Shared Memory Code .....	110
Debugging Portland Group, Inc., HPF Applications .....	111
Starting TotalView with HPF .....	113
Dynamically Loaded Library .....	114
Setting Up PGI HPF Compiler Defaults .....	115
Setting Up MPICH .....	115
Setting TotalView Defaults for HPF .....	115
Compiling HPF for Debugging .....	116
Starting HPF Programs .....	116
PGI HPF smp and rpm Libraries .....	116
Starting HPF Programs with MPICH .....	117
Workstation Clusters Using MPICH .....	117
IBM Parallel Environment.....	117
Parallel Debugging Tips .....	117
Attaching to Processes .....	117
General Parallel Debugging Tips .....	120
MPICH Debugging Tips .....	122
IBM PE Debugging Tips .....	122

## 6 Debugging Programs

Displaying Your Program's Call Tree .....	125
Finding the Source Code for Functions .....	127
Resolving Ambiguous Names .....	128

Finding the Source Code for Files .....	129
Examining Source and Assembler Code .....	129
Resetting the Current Stack Frame .....	132
Editing Source Text .....	132
Using the Toolbar to Select a Target .....	132
Stopping Processes and Threads .....	133
Updating Process Information .....	134
Holding and Releasing Processes and Threads .....	134
Examining Groups .....	135
Displaying Groups .....	137
Placing Processes into Groups .....	138
Starting Processes and Threads .....	138
Creating a Process Without Starting It .....	139
Creating a Process by Single-Stepping .....	139
Stepping .....	140
Process-Width Stepping .....	141
Group-Width Stepping .....	141
Thread-Width Stepping .....	142
Selecting Source Lines .....	142
Using Single-Step Commands .....	143
Stepping into Function Calls .....	144
Stepping Over Function Calls .....	144
Executing to a Selected Line .....	144
Executing to the Completion of a Function .....	146
Displaying Thread and Process Locations .....	147
Continuing with a Specific Signal .....	148
Setting the Program Counter .....	149
Deleting Programs .....	150
Restarting Programs .....	150
Checkpointing Programs and Processes .....	151
Interpreting Status and Control Registers .....	151

## 7 Examining and Changing Data

Displaying Variable Windows .....	153
Displaying Local Variables and Registers.....	153
Displaying a Global Variable .....	155
Displaying All Global Variables .....	155
Displaying Long Variable Names .....	155
Displaying Areas of Memory .....	156
Displaying Machine Instructions .....	157

Closing Variable Windows .....	158
Diving in Variable Windows .....	159
Changing the Values of Variables .....	161
Changing the Data Type of Variables .....	161
How TotalView Displays C Data Types .....	162
Pointers to Arrays .....	163
Arrays .....	163
Typedefs .....	164
Structures .....	164
Unions .....	165
Built-In Types .....	166
Character arrays (<string> Data Type) .....	168
Areas of memory (<void> Data Type) .....	168
Instructions (<code> Data Type) .....	169
Type Casting Examples .....	169
Displaying the argv Array .....	169
Displaying Declared Arrays .....	169
Displaying Allocated Arrays .....	170
Working with Opaque Data .....	171
Changing the Address of Variables .....	171
Changing Types to Display Machine Instructions .....	171
Displaying C++ Types .....	172
Classes .....	172
Changing Class Types in C++ .....	173
Displaying Fortran Types .....	174
Displaying Fortran Common Blocks .....	174
Displaying Fortran Module Data .....	175
Debugging Fortran 90 Modules .....	177
Fortran 90 User-Defined Type .....	178
Fortran 90 Deferred Shape Array Type .....	178
Fortran 90 Pointer Type .....	179
Displaying Fortran PARAMETERS .....	180
Displaying Thread Objects .....	181

## 8 Examining Arrays

Examining and Analyzing Arrays .....	183
Displaying Array Slices .....	183
Slice Definitions .....	184
Using Slices in the Variable Command .....	186
Array Data Filtering .....	188



Filtering by Comparison .....	188
Filtering for IEEE Values .....	189
Filtering by Range of Values .....	190
Array Filter Expressions .....	190
Filter Comparisons .....	192
Filtering Array Data .....	193
Sorting Array Data .....	193
Array Statistics .....	194
Displaying a Variable in All Processes or Threads .....	196
Diving in a Laminated Pane .....	198
Editing a Laminated Variable .....	198
Visualizing Array Data .....	199
Visualizing a Laminated Variable Window .....	200

## 9 Setting Action Points

Action Points Overview .....	201
Setting Breakpoints and Barriers .....	203
Setting Source-Level Breakpoints .....	203
Selecting Ambiguous Source Lines .....	204
Toggling Breakpoints at Locations .....	205
Ambiguous Locations .....	205
Displaying and Controlling Action Points .....	205
Disabling.....	207
Deleting .....	207
Enabling.....	207
Suppressing .....	207
Setting Machine-Level Breakpoints .....	208
Breakpoints for Multiple Processes .....	209
Breakpoint When Using fork()/execve() .....	211
Processes That Call fork() .....	211
Processes That Call execve() .....	211
Example: Multiprocess Breakpoint .....	212
Barrier Breakpoints .....	212
Barrier Breakpoint States .....	213
Setting a Barrier Breakpoint .....	213
Releasing Processes from Barrier Points .....	215
Deleting a Barrier Point .....	215
Changes When Setting and Clearing a Barrier Point .....	215
Defining Evaluation Points .....	216
Setting Evaluation Points .....	217

Creating Conditional Breakpoint Examples .....	218
Patching Programs .....	218
Conditionally Patching Out Code .....	219
Patching in a Function Call .....	219
Correcting Code .....	219
Interpreted vs. Compiled Expressions .....	220
Interpreted Expressions .....	220
Compiled Expressions .....	221
Allocating Patch Space for Compiled Expressions .....	221
Dynamic Patch Space Allocation .....	222
Static Patch Space Allocation .....	223
Using Watchpoints .....	224
Architectures .....	225
Creating Watchpoints .....	226
Displaying Watchpoints .....	227
Watching Memory .....	228
Triggering Watchpoints .....	228
Using Multiple Watchpoints .....	229
Data Copies .....	229
Conditional Watchpoints .....	230
Saving Action Points to a File .....	232
Evaluating Expressions .....	232
Writing Code Fragments .....	234
Intrinsic Variables .....	234
Built-In Statements .....	236
C Constructs Supported .....	238
Data Types and Declarations .....	238
Statements .....	239
Fortran Constructs Supported .....	239
Data Types and Declarations .....	240
Statements .....	240
Writing Assembler Code .....	241

## 10 Visualizing Data

How the Visualizer Works .....	247
Configuring TotalView to Launch the Visualizer .....	249
Data Types That TotalView Can Visualize .....	250
Visualizing Data from the Variable Window .....	251
Visualizing Data in Expressions .....	252
Visualizer Animation .....	254

Using the TotalView Visualizer .....	254
Directory Window .....	254
Data Windows .....	255
Viewing of Data .....	257
Graph Window .....	258
Displaying Graphs .....	258
Manipulating Graphs .....	260
Surface Window .....	260
Displaying Surface Data .....	262
Manipulating Surface Data .....	263
Launching the Visualizer from the Command Line .....	266
<b>11 Troubleshooting</b>	
Overview .....	269
The Problems .....	270
<b>12 X Resources</b>	
TotalView X Resources .....	275
Visualizer X Resources .....	284
<b>13 TotalView Command Syntax</b>	
Syntax .....	289
Options .....	290
<b>14 TotalView Debugger Server (tvdsvr) Command Syntax</b>	
The tvdsvr Command and Its Options .....	303
Replacement Characters .....	307
<b>A Compilers and Platforms</b>	
Compiling with Debugging Symbols .....	311
Compaq Tru64 UNIX .....	312
HP-UX .....	312
IBM AIX on RS/6000 Systems .....	313
SGI IRIX-MIPS Systems .....	314
SunOS 5 on SPARC .....	315
Using Exception Data on Compaq Tru64 UNIX .....	316
Linking with the dbfork Library .....	317
Compaq Tru64 UNIX .....	317
HP-UX .....	317
IBM AIX on RS/6000 Systems .....	318
Linking C++ Programs with dbfork .....	319

SGI IRIX6-MIPS .....	319
SunOS 5 SPARC .....	320

## **B Operating Systems**

Supported Operating Systems .....	321
Mounting the /proc File System .....	322
Compaq Tru64 UNIX and SunOS 5 .....	323
SGI IRIX .....	323
Swap Space .....	323
Compaq Tru64 UNIX .....	324
HP HP-UX .....	325
Maximum Data Size .....	325
IBM AIX .....	326
Linux .....	327
SGI IRIX .....	327
SunOS 5 .....	328
Shared Libraries .....	329
Changing Linkage Table Entries and LD_BIND_NOW .....	330
Using Shared Libraries on HP-UX .....	331
Debugging Dynamically Loaded Libraries .....	331
Known Limitations .....	334
Remapping Keys .....	334
Expression System .....	334
Compaq Alpha Tru64 UNIX .....	335
IBM AIX .....	335
SGI IRIX .....	335

## **C Architectures**

Compaq Alpha .....	337
Alpha General Registers .....	338
Alpha Floating-Point Registers .....	338
Alpha FPCR Register .....	339
HP PA-RISC .....	340
PA-RISC General Registers .....	340
PA-RISC Process Status Word .....	341
PA-RISC Floating-Point Registers .....	342
PA-RISC Floating-Point Format .....	343
IBM Power .....	344
Power General Registers .....	344
Power MSR Register .....	345

Power Floating-Point Registers .....	346
Power FPSCR Register .....	346
Using the Power FPSCR Register .....	348
Intel-x86 .....	348
Intel-x86 General Registers .....	349
Intel-x86 Floating-Point Registers .....	350
Intel-x86 FPCR Register .....	350
Using the Intel-x86 FPCR Register .....	351
Intel-x86 FPSR Register .....	352
SGI MIPS .....	352
MIPS General Registers .....	353
MIPS SR Register .....	354
MIPS Floating-Point Registers .....	355
MIPS FCSR Register .....	356
Using the MIPS FCSR Register .....	357
MIPS Delay Slot Instructions .....	357
Sun SPARC .....	358
SPARC General Registers .....	359
SPARC PSR Register .....	359
SPARC Floating-Point Registers .....	360
SPARC FPSR Register .....	361
Using the SPARC FPSR Register .....	362
<b>Glossary</b> .....	363
Citations .....	376
<b>Index</b> .....	377





## About This Book



This guide describes how to use TotalView®, a source-level and machine-level debugger with an easy-to-use interface and support for debugging multiprocess programs. The guide assumes that you are familiar with programming languages, the UNIX operating systems, the X Window System, and the processor architecture of the platform on which you are running TotalView.

This guide covers using TotalView on any platform. Most of the examples and illustrations in this guide show TotalView running on a workstation. To learn about the specifics of running TotalView on your platform, refer to Appendix A, “Compilers and Platforms,” on page 311, Appendix B, “Operating Systems,” on page 321, and Appendix C, “Architectures,” on page 337.

## Supported Platforms



TotalView runs on a variety of platforms and can be used to debug programs running locally or on remote systems. It can debug parallel processors, supercomputers, and digital signal processor boards.

If TotalView is not yet available for your system configuration, please contact Etnus® about porting TotalView to suit your needs:

**Etnus Inc.**  
24 Prime Parkway  
Natick, MA 01760  
Internet E-mail: [info@etnus.com](mailto:info@etnus.com)  
1-800-856-3766 in the United States  
(+1) 508-652-7700 worldwide

## Reporting Problems

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Internet E-Mail addresses: **support@etnus.com**

United States Phone Number: 1-800-856-3766

Worldwide Phone Number: (+1) 508-652-7700

If you are reporting a problem, please include the following information:

- The **version** of TotalView
- The **platform** on which you are running TotalView
- An **example** that illustrates the problem
- A **record** of the sequence of events that led to the problem

See the TOTALVIEW RELEASE NOTES for complete instructions on how to report problems.

## Conventions

The following table describes the conventions used in this book:

Table I: Book Conventions

Convention	Meaning
[ ]	Brackets are used when describing parts of a command that are optional.
<i>arguments</i>	Within a command description, text in italic represent information you type. Elsewhere, italic is used for emphasis. You will not have any problems distinguishing between the uses.
<b>Dark text</b>	Within a command description, <b>dark text</b> represent key words or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.



# Chapter 1

## TotalView Features

The Etnus TotalView® debugger is a sophisticated tool that allows you to debug, analyze, and tune the performance of complex multiprocessor or multi-threaded programs.

This chapter highlights:

- TotalView Advantages
- TotalView Windows
- Understanding Multiprocess Programs
- Understanding Multithreaded Programs
- Controlling Processes and Threads
- Using Action Points
- Examining and Manipulating Data
- Distributed Debugging
- Visualization
- Context-Sensitive Help and Documentation

If you want to jump in and get started quickly, you should go to our Website at [www.etnus.com](http://www.etnus.com) and go to TotalView's "Getting Started" area.

## TotalView Advantages

TotalView provides many advantages over conventional UNIX debuggers such as **dbx**, **gdb**, **adb**, and other sophisticated hardware-specific debuggers:

- TotalView runs on all major UNIX platforms so you can use the same debugger regardless of where you are debugging today.

- TotalView's interface lets you see a lot of useful information without entering commands.
- You can debug *multiprocess multithreaded* programs. TotalView displays all important information about a single process in its own window, showing the source code, stack trace, and stack frame for one or more threads in the process. By default, TotalView displays just the current process, but you can display all process windows simultaneously if you wish. TotalView even lets you perform debugging tasks across processes.
- TotalView's distributed architecture lets you debug *remote* programs over the network, as shown in the following figure.

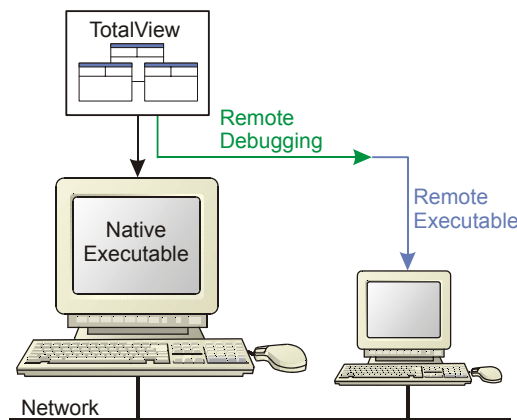


FIGURE 1: **Debugging a Remote Program with TotalView**

TotalView can manage multiple remote programs and multiprocess multithreaded programs simultaneously, as shown in Figure 2.

- Parallel and distributed programs run in many processes, and your debugger must know about them. When you start TotalView as part of an MPI, IBM Parallel Environment (PE), OpenMP, pthread, HPF, or Parallel Virtual Machine (PVM), application, TotalView automatically detects and attaches to these processes. If a program calls **fork()** or **execve()**, TotalView automatically attaches to the child process. This is called *automatic process acquisition*.
- Because your program can execute using a variety of processes and threads, TotalView has an extremely flexible system of handling your executable so that you can control how and what is executing at any one time.

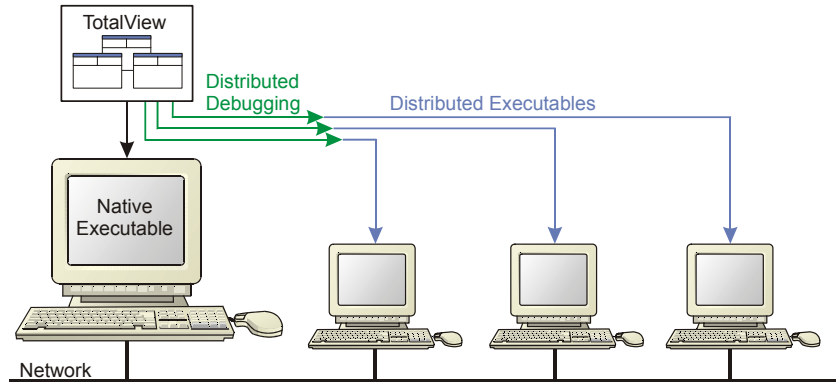


FIGURE 2: **Debugging a Distributed Program with TotalView**

- Because TotalView lets you attach to running processes, you can debug processes that were not started under TotalView's control.
- If the code you are debugging was not compiled using the `-g` option or if you do not have access to the program's source file, TotalView lets you debug its machine-level code.
- TotalView lets you temporarily add source code statements to the program you are debugging. On some platforms, you can even add machine code statements. This feature saves time when you are testing bug fixes.
- TotalView's Command Line Interface (CLI) lets you enter commands directly in an xterm window when you find yourself unable to use the GUI. (The CLI is described in the CLI GUIDE.)

## TotalView Windows

The three most often used windows within TotalView are shown in Figure 3.

- |                    |  |
|--------------------|--|
| <b>Root Window</b> | <p>Gives you an overview of the state of your program. You can also use it as a navigation tool.</p> <p>This window has tabbed pages that contain information about the processes and threads being debugged, processes that TotalView could acquire for debugging, groups that you can use to manipulate these processes and threads, and a log that records your entire session.</p> |
|--------------------|--|

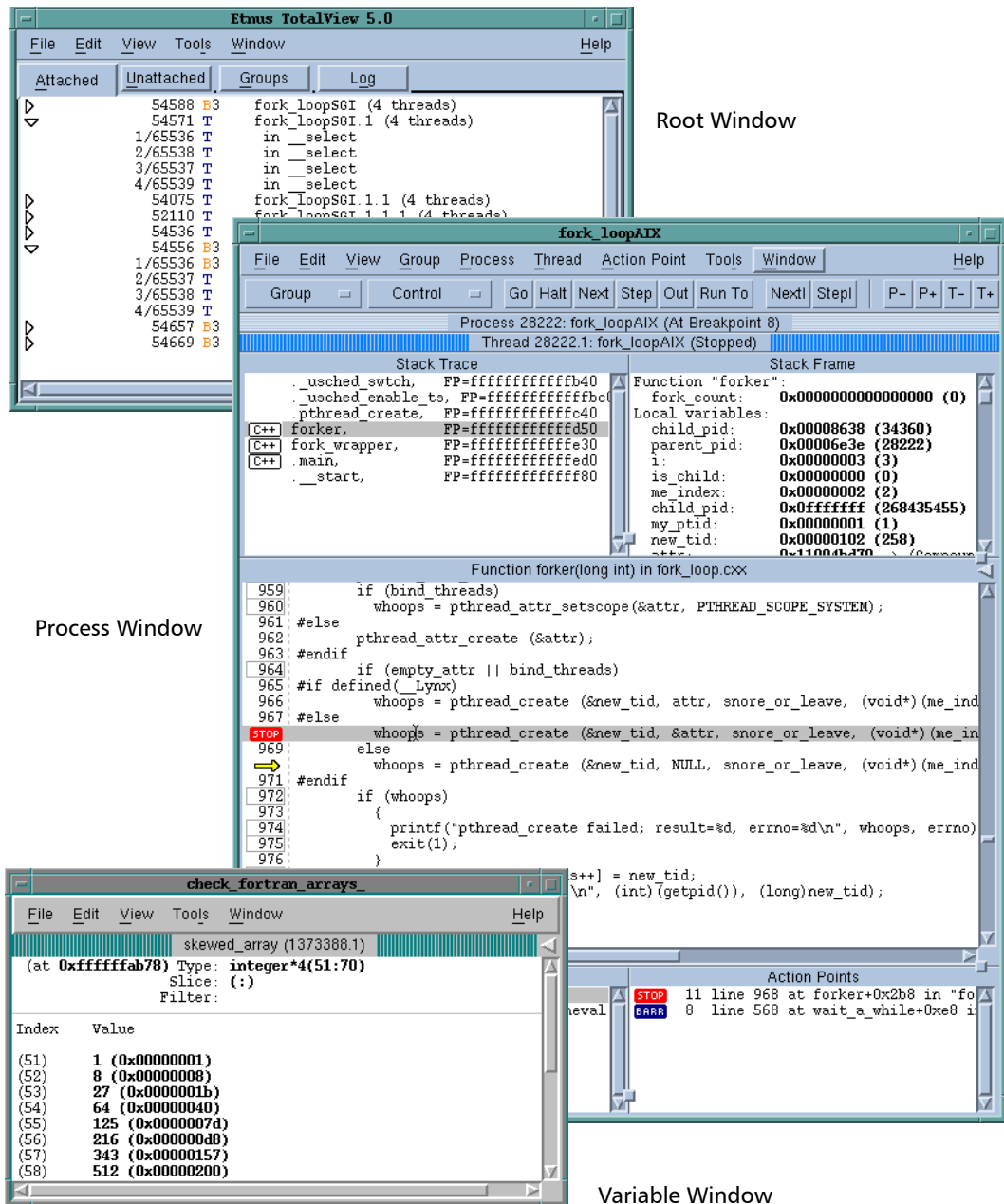


FIGURE 3: TotalView Windows

**Process Window** Displays information about a process and a thread within that process. Panes within this window show the stack trace, stack frame, and code for the selected thread. This window is where you will spend the bulk of all your debugging activities.

**Variable Window** Lists the address, data type, and value of a local variable, register, or global variable. It also shows the values stored in a block of memory.

## Understanding Multiprocess Programs

TotalView has special features for debugging multiprocess programs.

### ■ Process groups

TotalView treats multiprocess programs as groups of processes. When debugging multiprocess programs, you can view information about all process groups and can view information about a multiprocess program. Using TotalView, you can start and stop individual groups.

### ■ Separate windows for each process

TotalView can display each process in its own TotalView Process Window, displaying information for that process. You can monitor the status, thread list, breakpoint list, and source code for *each* process. You do not have to display all the process windows in a multiprocess program; instead, you can choose only those Process Windows that are important to you.

Because display space is always limited on your monitor, TotalView tries to reuse Process Windows unless you tell it not to. In this way, you are not forced to have hundreds of windows when you are working with hundreds of processes.

### ■ Sharing of breakpoints among processes

You can control if a breakpoint is shared among child processes and if all processes in the group stop when any process in the group reaches a breakpoint. Like most behaviors in TotalView, you can set preferences or use command-line options that configure how TotalView behaves.

### ■ Barrier breakpoints

In addition to "normal" breakpoints, TotalView allows you to create process and thread barrier breakpoints. A barrier breakpoint differs from a

normal breakpoint in that it holds every thread or process in a group that reaches the barrier until all reach it. When the last in the group reaches the barrier, TotalView releases all of these held processes. This lets you synchronize a group of processes or threads to the same location.

#### ■ **Process and thread group-level single-stepping**

TotalView allows you to single-step groups of processes and groups of threads using one command.

#### ■ **Multiple symbol tables**

If you are debugging more than one executable at the same time, TotalView automatically handles the symbol table for each.

## Understanding Multithreaded Programs

While the way in which operating systems implement threads vary, most share the following characteristics:

#### ■ **Shared address space**

The threads share an address space (memory) with other threads. They can read and write the same variables and can execute the same code.

#### ■ **Private execution context**

Each thread has its own general-purpose and floating-point registers.

#### ■ **Thread private data**

Some operating systems allow a program to declare thread private data. This declaration provides each thread with its own copy of the variable. Changes made by one thread to its private variables are not seen by other threads.

#### ■ **Private execution stack**

Each thread has an address space reserved for its execution stack. However, one thread's stack can be read and written by other threads sharing the address space.

TotalView can help you debug threaded applications on a variety of operating systems. On some of these systems, a process consists of an address space and a list of one or more threads. Other operating systems implement tasks or threads running in the computer's memory space and do not support multiple processes or address spaces on a single machine.

Because the ways operating systems handle threads differ, TotalView implements a general model of address spaces and execution contexts. A TotalView *thread* refers to a thread or task with an execution context, and *process* refers to an address space or computer memory that can run one or more threads.

## Controlling Processes and Threads

TotalView offers a full range of methods for controlling processes and threads. Using TotalView, you can:

- **Automatically attach to processes**

When your program creates processes and threads on your current computer or another, TotalView automatically attaches to them, making their symbol tables available to you and allowing you to manipulate them in the same way as you manipulate the process originally started under TotalView's control.

- **Automatically create groups**

When processes and threads are created, TotalView automatically adds them to groups. TotalView places every process and every thread into two or more groups that can be manipulated using group, process, and thread commands. Using the CLI (which is TotalView's Command Line Interface), you can create groups that can be manipulated from the TotalView Process Window. For example, you can step all threads in a group without stepping other threads in their processes.

- **Start and stop processes and threads**

You can start, stop, resume, delete, restart, and even reload recompiled versions of your program.

- **Attach to existing processes**

TotalView lets you examine processes that are not yet running under its control. Attaching to one of these processes is as easy as double-clicking on the process's name in the Root Window.

- **Examine core files**

You can load a core file and examine it in the same way as any other executable. Or, you can load a core file at anytime.

### ■ **Single-step your program**

You can single step through your program or step over function calls. You can tell your program to execute to a selected source line or instruction, or continue executing until a function completes its execution. TotalView supports process-level, process group-level, and, on some systems, thread-level single stepping.

### ■ **Change the way TotalView handles signals**

You can indicate how TotalView handles signals. For example, it can stop the process and place it in a stopped or error state, sending the signal on to the process, or discarding the signal.

## Using Action Points

TotalView provides a broad range of action points. (Action points are places in a program where you stop execution or evaluate an expression.)

**Action points:** You can set, delete, suppress, unsuppress, enable, and disable action points at the source and machine levels. TotalView lets you set the following action points:

- **Breakpoints** stop execution when a statement or instruction executes.
- **Barrier breakpoints** hold other threads until all threads in a group reach a “barrier” statement or instruction.
- **Conditional breakpoints** only perform an action if a code fragment (expression) is satisfied.
- **Evaluation points** execute code you create at a statement or instruction.
- **Watchpoints** monitor when changes occur to a variable’s value.

**Expressions and code fragments:** TotalView lets you write and evaluate code fragments, including function calls used by the current process. While differences exist between platforms, you can write fragments in C, C++, Fortran, and assembler. On most platforms, TotalView compiles code fragments. This is a great way to test a fix without altering your source and recompiling it.



## Examining and Manipulating Data

TotalView provides many ways for you to examine your code.

### ■ Diving

You can obtain additional information about almost everything that is displayed in TotalView by clicking on it. This process, which is called *diving*, tells TotalView that it should display the selected information in some way.

For example, if you double-click on a function name (or use the **View > Dive** command), TotalView displays the function's source code in the Process Window's Source Pane.

You can dive into a variable in the same way that you dive into a function. That is, either double-click on the variable name or select **View > Dive** while the cursor is over the variable. TotalView lets you examine local variables, registers, global variables, machine-level instructions, and areas of memory.

You can dive upon almost everything that you see in a TotalView window.

### ■ Search for functions

TotalView's **View > Lookup Function** command lets you search for functions.

### ■ Change a variable's type and value—Casting and Type Transformation

You can alter a variable's type to display the data in different formats and you can edit a variable value or a memory location, changing it for the current running process.

Using the CLI, you can tell TotalView that it should display information in the way you want it displayed. For example, you can display information in a C++ STL as if it were a normal structure or array. (Information on "type transformation" can be found in the CLI GUIDE.

### ■ Laminate variables

You can examine the value of a variable across multiple processes and multiple threads in a single Variable Window. (This ability to display the multiple values of a variable is called *lamination*.)

### ■ Examine array data

You can filter array data to look for elements that match a filter expression. You can also sort data and tell TotalView to display statistical information about an array's contents.

## Distributed Debugging

TotalView provides a distributed architecture that supports many different operating environments, including:

- Remote programs running on a separate machine from TotalView.
- Multiprocess programs running on a multiprocessor machine.
- Multiprocess programs running on a cluster of homogeneous machines.
- Distributed programs running on a set of homogeneous machines.

**NOTE** Distributed debugging requires that all machines have the same architecture and operating system.

The machine on which TotalView is running is known as the *host machine*, while the machine on which the process being debugged is running is the *target machine*. The host and target machines can be the same machine.

If the host and target machines are different, TotalView starts a process on each remote target machine. TotalView communicates with this process by using standard TCP/IP protocols. (See Figure 4.)

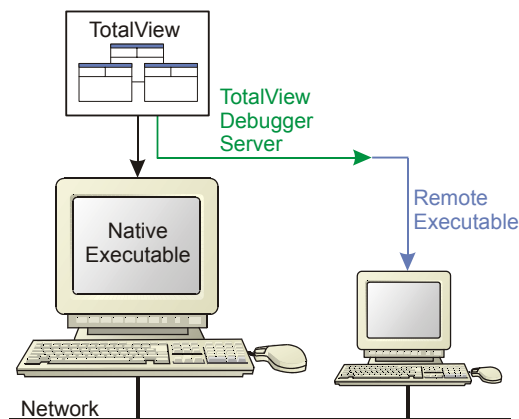


FIGURE 4: TotalView Debugger Server

Debugging distributed programs does not differ from debugging nondistributed programs: TotalView offers the same set of features for each.

Depending on the platform, TotalView can debug programs that use the HPF, MPI, IBM Parallel Environment (PE), OpenMP, pthreads, and Parallel Virtual Machine (PVM) libraries.

## Visualization

TotalView gives you a variety of ways to visualize your data.

- The TotalView Visualizer allows you to graphically view array data in the programs you are debugging. This gives you an overall picture of your data and helps you find incorrect data quickly and easily.

**NOTE** The Visualizer is not available on Linux Alpha and 32-bit SGI Irix.

Each time you visualize the same array, the Visualizer image is updated. See Figure 5:

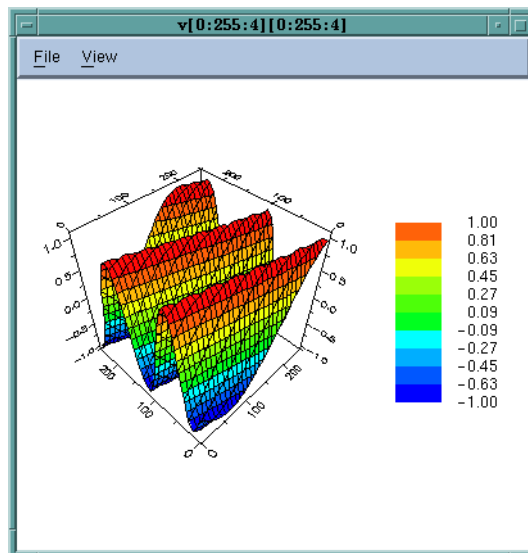


FIGURE 5: **Sample Array Visualization**

- TotalView's **Call Tree** (see Figure 6) lets you see the sequence of calls associated with any current routine. This display is dynamic as it displays a program's call tree at the time when you ask for it.

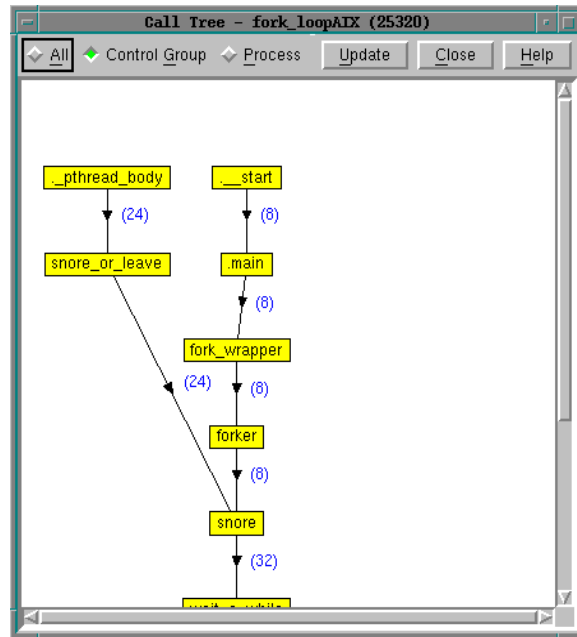


FIGURE 6: **Sample Call Tree**

- TotalView's **Message Queue Graph** visually displays the processes (as rank numbers) that are linked together by the messages that each send and receive. You can control what you are seeing by selecting Pending Sends, Pending Receives, and Unexpected Messages. Another control lets you select which messages participate in the graph. (See Figure 7.)

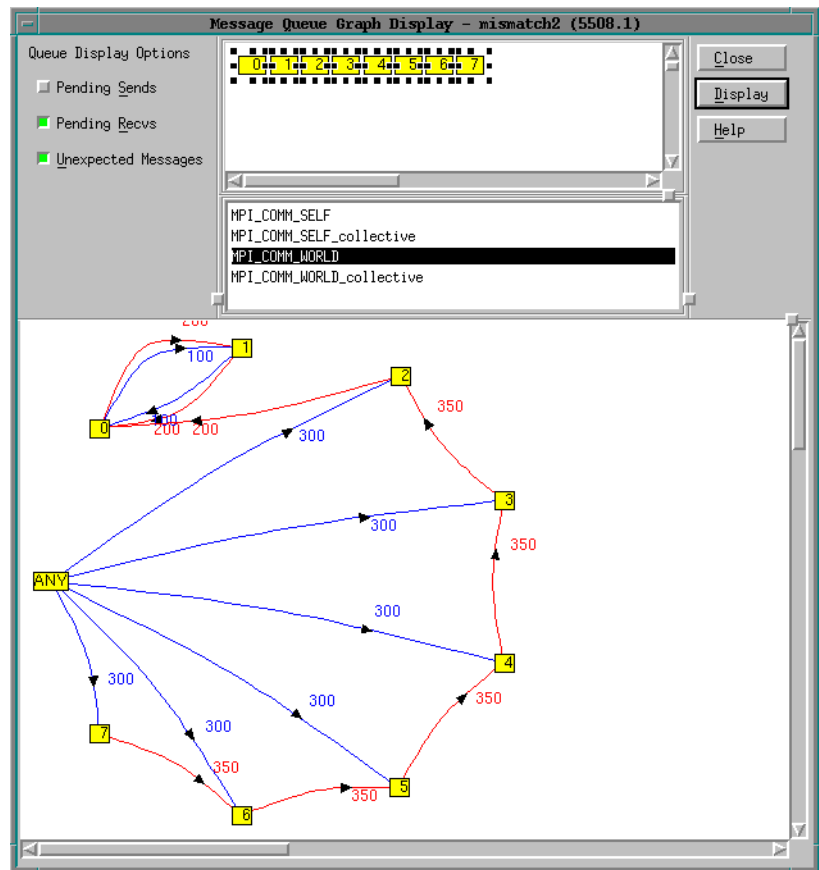


FIGURE 7: Sample Message Queue Graph

## Context-Sensitive Help and Documentation

You can request help from any window being displayed. The **Help** command displays context-sensitive information about the current window or dialog box or the debugging operation you are currently using. TotalView displays the information in a separate help window.

**NOTE** Context-sensitive help is not available on Linux Alpha platforms.

Also contained within the help is the full text of all TotalView documents. HTML and PDF versions of our documents are located on our Web site, which is [www.etnus.com](http://www.etnus.com). Some of our customers also install this information locally.

Most documentation is available in printed form. Information about this documentation is on our Web site.

# TotalView Basics

This chapter introduces you to the TotalView interface and describes:

- Compiling Programs
- Starting TotalView
- Using the Mouse Buttons
- Using the Root Window
- The Process Window
- Diving into Objects
- Editing Text
- Searching for Text
- Searching for Functions and Variables
- Saving the Contents of Windows
- Exiting from TotalView

## Compiling Programs

Before starting TotalView, compile your source code with the `-g` compiler option. This option tells your compiler to generate symbol table debugging information. For example:

```
cc -g -o executable source_program
```

For more information on compiling your program for TotalView, see “*Compiling Programs*” on page 33.

On some platforms, you may need to use additional compiler options. Refer to Appendix A, “*Compilers and Platforms*” on page 311 for more information.

TotalView also lets you debug programs that were not compiled with the `-g` option or programs for which you do not have source code. For more information, refer to “*Examining Source and Assembler Code*” on page 129.

When TotalView reads a file, it uses the file’s extension to determine the programming language that you used to write the file’s contents, as shown in the following table.

TABLE 1: Source Language Mapping

File Extension	Source Language
.cxx, .cc, .cpp, .C, .hxx, .H	C++
.F, .f, .F90, .f90	FORTRAN 77 or Fortran 90
.hpf, .HPF	HPF
All others	C

TotalView identifies a program as FORTRAN 77 or Fortran 90 when:

- The compiler’s debugging information includes the programming language that you used.
- The source file name has an `.f90` or `.F90` suffix.
- The code uses Fortran 90 features such as assumed shape arrays or pointers.

If TotalView cannot identify a source file’s language, it assumes that the source language is C. If this causes problems, you will need to change the file’s extension to one that TotalView recognizes.

## Starting TotalView

Depending on the kind of program you are debugging, there are several ways to start TotalView. The simplest method uses the **totalview** command and your program’s name:

```
totalview executable
```

A similar command can be used to start the CLI:

```
totalviewcli executable
```



You can also invoke the CLI by selecting the **Tools > Command Line** command. The CLI is described in the CLI GUIDE.

In many cases, the way you will start your program requires that you define an environment, pass in command-line arguments, set the number of processes, and so on. You will find this information in the following places:

- "Starting TotalView" on page 34.
- "Setting Up Parallel Debugging Sessions" on page 77.

See Chapter 13, "TotalView Command Syntax" on page 289 for complete information on the **totalview** command.

## Initializing the Debugger

An *initialization file* contains commands that let you modify the TotalView and CLI environments and add your own functions to this environment. TotalView allows you to place information in more than one file. These files can be located in your installation directory, your home directory, or the directory from which you invoked TotalView. If it is present in one of these places, TotalView reads and executes its contents.

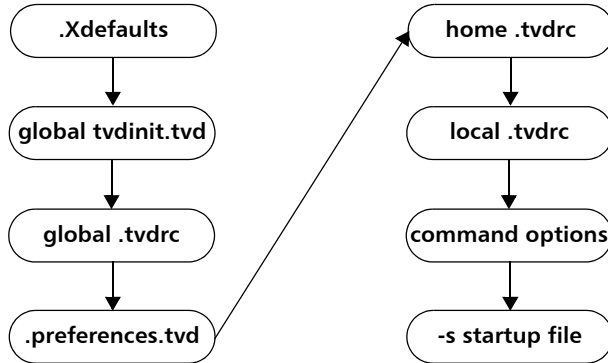
Typically, **.tvdrc** files contain command, function, variable definitions, and function calls that you want executed whenever you start a new debugging session.

If you add the **-s filename** option to either the **totalview** or **totalviewcli** shell commands, you can have TotalView execute the CLI commands contained within *filename*. Your startup file executes after **.tvdrc** files execute.

The following figure shows the order in which initialization and startup files execute:

The **-s** option lets you, for example, initialize the debugging state of your program, run the program you are debugging until it reaches some point where you are ready to begin debugging, and even lets you create a shell command that starts the CLI.

**NOTE** The **.Xdefaults** file, which is actually read by the server when you start X Windows, is only used by the GUI. The CLI ignores it.

FIGURE 8: **Startup and Initialization Sequence**

As part of the initialization process, TotalView exports two environment variables into your environment: `LM_LICENSE_FILE` and either `SHLIB_PATH` or `LD_LIBRARY_PATH`.

If you have saved a breakpoint file into the same subdirectory as your program, TotalView automatically reads the information in this file when it loads your program.

**NOTE** The format of a Release 5.0 breakpoint file differs from that used in earlier releases. While Release 5 versions of TotalView can read breakpoint files created by earlier versions, earlier versions cannot read a Release 5 breakpoint file.

## Using the Mouse Buttons

TotalView uses the buttons on your three-button mouse as follows:

TABLE 2: **Mouse Button Functions**

Button	Action	Purpose	How to Use It
Left	Select	Selects or edits object, scrolls in windows and panes	Move the cursor over the object and click the button.

TABLE 2: Mouse Button Functions (Continued)

Button	Action	Purpose	How to Use It
Middle	Paste	Writes information previously copied or cut into the clipboard	Move the cursor to where you will be inserting the information and click the button; not all windows support pasting.
Right	Context menu	Displays a menu with commonly used commands	Move the cursor over an object and click the button.  Most windows and panes have context menus; dialog boxes do not have context menus.

In most cases, a single-click selects what is under the cursor and a double-click dives on the object. However, if the field is editable, TotalView goes into its edit mode where you can alter the selected item's value.

In some locations such as the Stack Trace Pane, selecting a line tells TotalView that it should perform an action. In this case, TotalView dives on the selected routine. (In this case, *diving* means that TotalView finds the selected routine and show it in the Source Pane.)

In the tag field area (the area on the left containing source code numbers) of the Source Pane, the left button sets a breakpoint at that line. TotalView shows you that it has set a breakpoint by displaying a **STOP** icon in the tag field.

Selecting the **STOP** icon a second time deletes the breakpoint. If, however, you had created an evaluation or event point—this is indicated by an **EVAL** icon—selecting the icon disables it. For more information on breakpoints and evaluation points, refer to Chapter 9, “*Setting Action Points*” on page 201.

## Using the Root Window

The Root Window appears when you start TotalView. If you do not specify a program name when starting TotalView, it is the only window that appears. If you indicate a program name, TotalView will also open a Process Window for the program.

The *Root Window* contains four pages, as follows:

- **Attached:** Displays a list of all the processes and threads being debugged. Initially, the Root Window just contains the name of the program being debugged. Associated with each is a name, location (if a remote process), process ID, status, and a list of threads for each process being debugged if it has begun executing. It also shows the thread ID, status, and current routine executing for each thread.

Figure 9 shows the Attached Page for an executing multithreaded multi-process program.

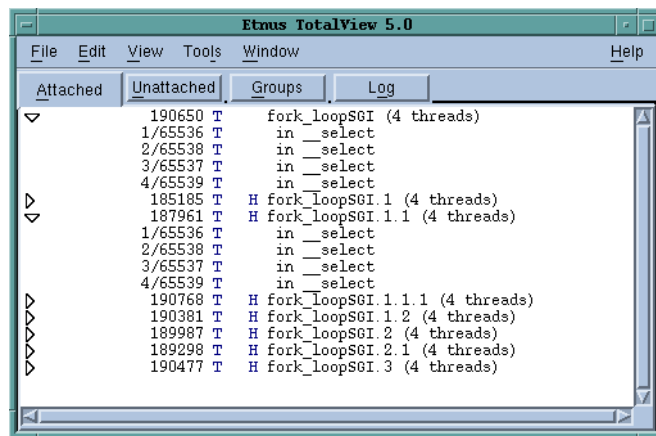


FIGURE 9: Root Window Attached Page

- **Unattached:** Displays processes over which you have control. If you cannot attach to one of these processes—for example, you cannot attach to the TotalView process—TotalView displays it in gray. Figure 10 shows the Unattached Page.
- **Groups:** Lists the groups used by your program. The top pane lists all of your program's groups. This list includes all the groups that TotalView creates and all that you create using the CLI. When you select a group in the top pane, the group's members are displayed in the bottom pane. Figure 11 shows a Groups Page.

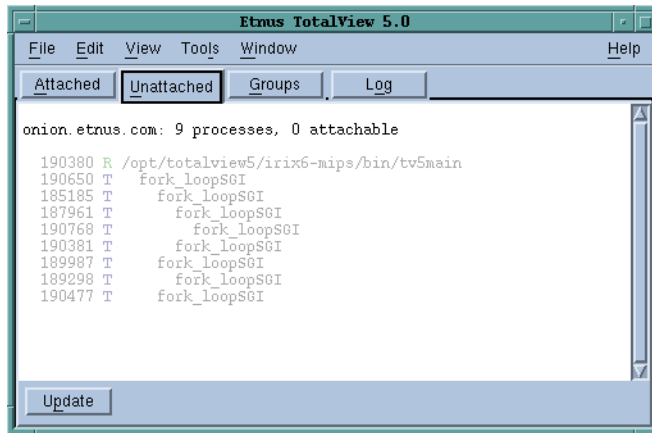


FIGURE 10: Root Window Unattached Page

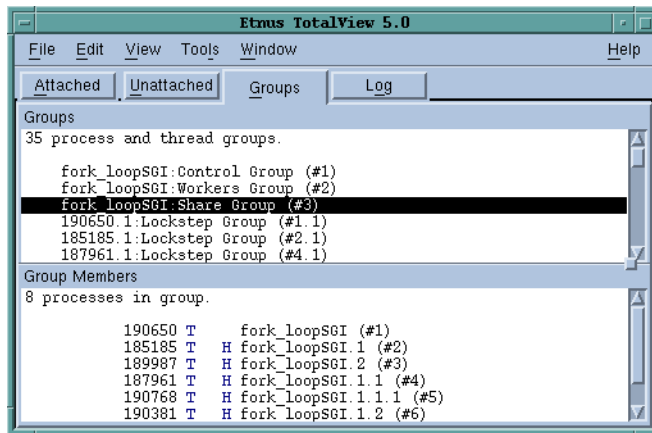


FIGURE 11: Root Window Groups Page

**NOTE** Until you are comfortable using TotalView's group model, you are strongly urged to investigate group membership using this window. This window gives you considerable insight into how TotalView manipulates groups and what TotalView causes to run when you execute commands such as *step* or *go*.

- **Log:** Contains a log of debugging information.

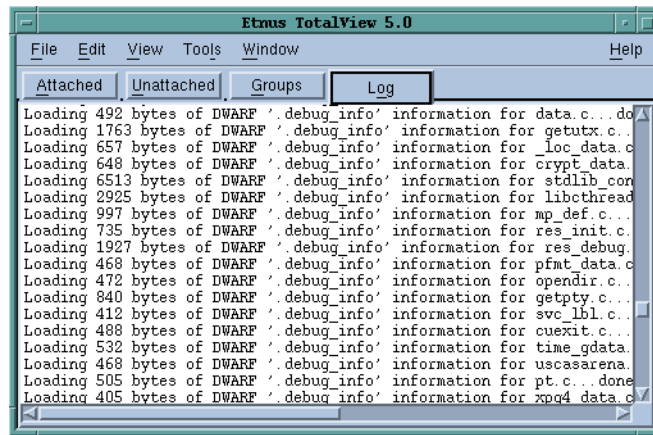


FIGURE 12: Root Window Log Page

## The Process Window

The *Process Window* displays five *panes* of information. (The contents of these panes are discussed later in this section.) The large scrolling list in the middle of the Process Window is the Source Pane. As its name suggests, this pane initially contains your source code.

Figure 13 shows the Process Window.

## Starting a Process

In many cases, the way you will start a process is as follows:

- 1 Set a breakpoint in the source code by selecting a boxed line number.
- 2 Type the keyboard accelerator **g** (for the **Process > Go** command).  
The process starts running and then stops at the first breakpoint set.

When debugging a remote process, TotalView displays an abbreviated version of the host name on which the process is running within brackets ([ ]) in the Root Window. The full host name appears in brackets in the title bar

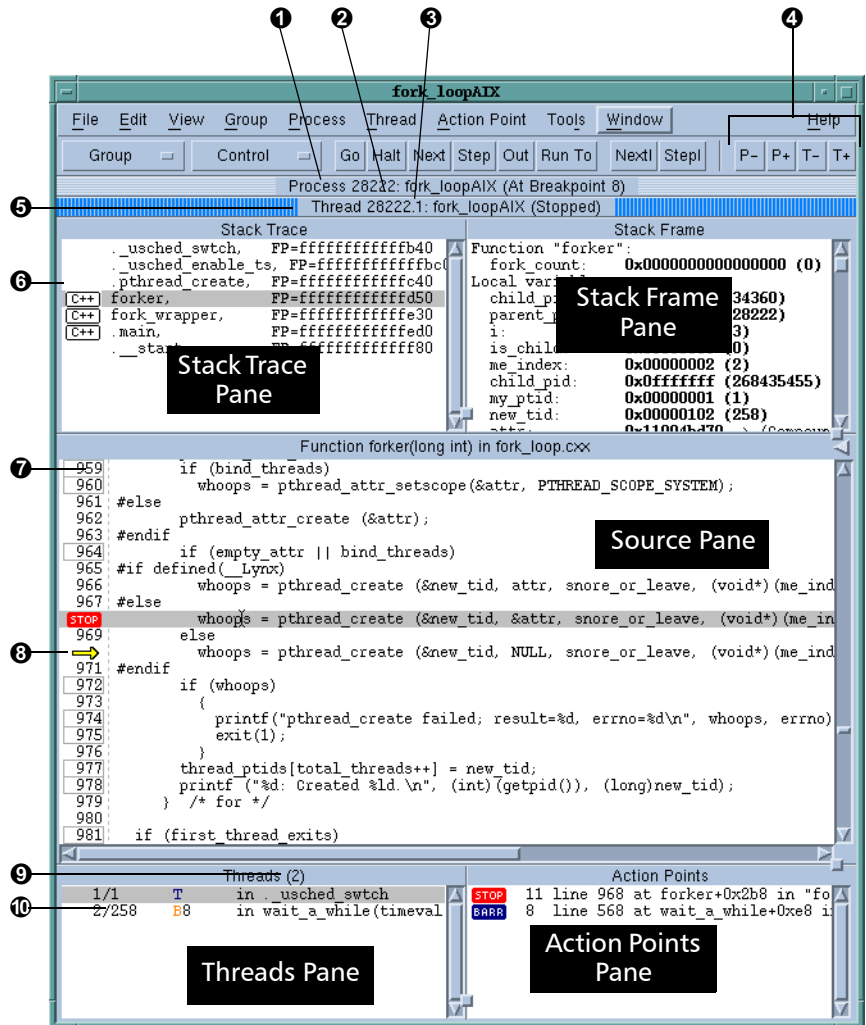
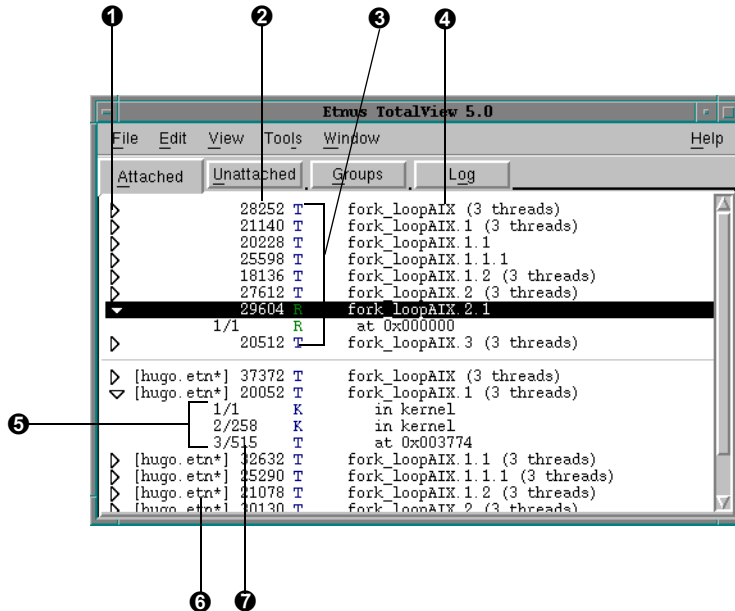


FIGURE 13: Process Window

of the Process Window. In Figure 14, the process is running on the machine `hugo.etnus.com`, which is abbreviated to `[hugo.etn*]` in the Root Window.



- ❶ Collapse/expand toggle
- ❷ Process ID (PID)
- ❸ Thread status
- ❹ Program name
- ❺ Thread list
- ❻ Remote process location
- ❼ Thread ID (TID/SYSTID)

FIGURE 14: Root Window Showing Remote

As you examine the Process Window in Figure 13 (on the previous page), notice the following:

- The thread ID shown in the Root Window and in the process's Threads Pane is the TotalView assigned logical thread ID (or TID) and system assigned thread ID (or SYSTID). On systems such as Compaq Tru64 UNIX where the TID and SYSTID values are the same, TotalView displays only the TID value.

In other windows, TotalView uses the value `pid.tid` to identify a process's threads.

The *Threads Pane* shows the list of threads that currently exist in the process. The number in the Threads Pane title (❹) is the number of threads



that currently exist in the process. When you select a different thread in this list, TotalView updates the Stack Trace Pane, Stack Frame Pane, and Source Pane to show the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window displaying information for that thread.

- The *Stack Trace Pane* shows the call stack of routines that the selected thread is executing. You can move up and down the call stack by selecting the routine (stack frame). When you select a different stack frame, TotalView updates the Stack Frame and Source Panes to show the information about the selected routine.
- The *Stack Frame Pane* displays all the function parameters, local variables, and registers for the selected stack frame.
- The information displayed in the Stack Trace and Stack Frame Panes reflects the state of the process when it was last stopped. Consequently, this information is not up-to-date while the thread is running.
- The left margin of the *Source Pane*—called the tag field area—displays line numbers and icons indicating something about your program. You can place a breakpoint at any source code line that displays object code. (These places are indicated by a boxed line number.) When you place a breakpoint on a line, TotalView overwrites the line number with a STOP icon. The arrow in the tag field shows the current location of the program counter (PC) within the selected stack frame. See Figure 15.

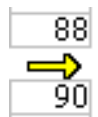


FIGURE 15: **Process Window Tag Field Area**

- **Point of execution.** This means that each thread's Process Window has its own unique program counter (PC). When you stop a multiprocess or multithreaded program, the routine selected in the Stack Trace Pane for a thread depends on the thread's PC. When you stop the program, some threads can be executing in one routine, while others might be executing elsewhere.
- The *Action Points Pane* shows the list of breakpoints, evaluation points, and watchpoints for the process.

## Diving into Objects

One of the main functions of the Root Window is quickly display a Process Window that contains information about the selected processes and threads. The procedure is simple: you select what you want to see and then double-click on it. In most cases, you can display more detail about something by placing the cursor over a name or object and diving into it by double-clicking the left mouse button.

**NOTE** In some cases, single-clicking tells TotalView to dive. For example, diving on a function name in the Stack Trace Pane tells TotalView to dive into the function.

Table 3 describes some of the information you can dive on.

TABLE 3: Diving

Dive on:	Information Displayed by Diving:
Process or thread	When you dive on a processor thread in the Root Window, TotalView finds or opens a Process Window for that process. If it cannot find a matching window, TotalView replaces the contents of an existing Process Window and shows you the selected process.
Subroutine	The source code for the subroutine replaces the current contents of the Process Window—this is called a <i>nested dive</i> . When this occurs TotalView places a right angle bracket (>) in the process's title. Every time it dives, it adds another angle bracket.

```
Function >>>>fork_wrapper(int) in fork_loop.cxx
```

FIGURE 16: Nested Dive

A subroutine must be compiled with source-line information (usually, with the **-g** option) for you to dive into it and see source code. If the subroutine was not compiled with this information, TotalView displays the routine's assembler code.

TABLE 3: Diving (Continued)

Dive on:	Information Displayed by Diving:
Pointer	The referenced memory area appears in a separate Variable Window.
Variable	The contents of the variable appear in a separate Variable Window.
Array element, structure element, or referenced memory area	The contents of the element or memory area replaces the contents that were in the Variable Window—this is known as a <i>nested</i> dive.
Routine in the Stack Trace Pane	The stack frame and source code for the routine appear in a Process Window.

TotalView tries to reuse windows whenever possible. For example, if you dive on a variable and that variable is already being displayed in a window, TotalView pops the window to the top of the display. If you want the information to appear in a separate window, use the Root Window's **View > Dive Anew** command.

**NOTE** Using **View > Dive Anew** on a process or a thread may not create a new window if TotalView determines that it can reuse a Process Window. If you really want to see the information in two windows, use the Process Window's **Window > Duplicate** command.

For additional information about displaying variable contents, refer to “*Diving in Variable Windows*” on page 159.

Other windowing commands that you can use are:

- **Window > Duplicate:** (Variable Window) Creates a duplicate copy of the current Variable Window.
- **Window > Duplicate Base:** (Variable Window) Creates a duplicate copy of the current Variable Window. In contrast with **Window > Duplicate**, this command contains the dive stack.
- **File > Close:** Closes an open window.
- **File > Close Relatives:** Closes windows that are related to the current window but does not close the current window.
- **File > Close Similar:** Closes the currently open window and all windows similar to it. When you have lots of similar windows, this is a great time-saver.

## Editing Text

The TotalView field editor lets you change the values of fields in windows or change text fields in dialog boxes. To edit text:

- 1 Click the left mouse button to select the text you wish to change. If you can edit the selected text, it appears within a highlighted rectangle, and you will see an editing cursor.

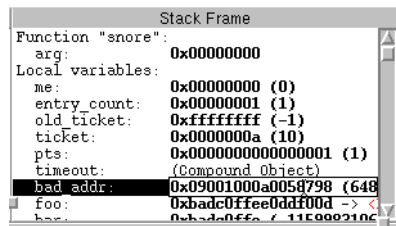


FIGURE 17: Editing Cursor

- 2 Edit the text and press Return.

Like other Motif-based applications, you can use your mouse to copy and paste text within TotalView and to other X-based applications by using your mouse buttons.

You can also manipulate text by using **Edit > Copy**, **Edit > Cut**, **Edit > Paste** and **Edit > Delete**.

## Searching for Text

You can search for text strings in most windows using the **Edit > Find** command. After invoking this command, TotalView displays the dialog box shown in Figure 18.

The commands within this dialog box let you search **Down** (which is towards the end of the current window) or **Up** (which is the other way). Selecting the **Case Sensitive** button tells TotalView that it should only locate text having the same capitalization as the text entered in the **Find** field.

After you have found a string, you can reexecute the command by using the **Edit > Find Again** command.

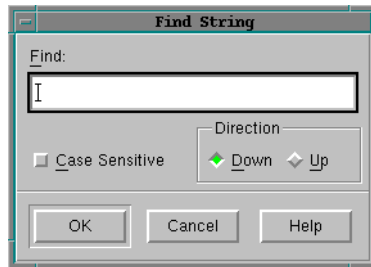


FIGURE 18: **Edit > Find Dialog Box**

## Searching for Functions and Variables

In many cases, having TotalView locate a variable or a function is much easier than scrolling through your sources looking it. The **View > Lookup Function** and **View > Lookup Variable** commands, which display the dialog box displayed in Figure 19, let TotalView find them for you.

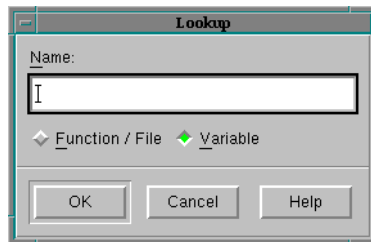
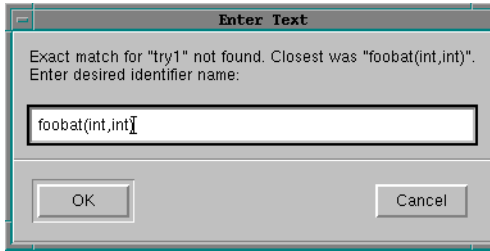


FIGURE 19: **View > Lookup Variable Dialog Box**

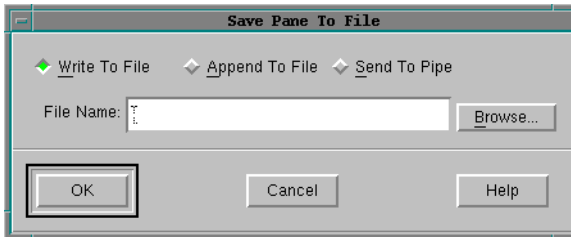
If TotalView does not find the name you entered, it displays a dialog box with the closest match, as shown in Figure 20.

If you select the OK button and TotalView cannot find the function or variable, it displays a dialog box saying that it found a similar object. If you like this object, select **OK** to have it displayed in the Source Pane.

FIGURE 20: **Spelling Corrector Dialog Box**

## Saving the Contents of Windows

You can save the contents of most pages and panes as ASCII text with the **File > Save Pane** command. You can also use this command to pipe data to a UNIX shell command.

FIGURE 21: **File > Save Pane Dialog Box**

When piping information, TotalView pipes the commands to `/bin/sh` for execution. This means that you can use a series of shell commands. For example, here is a command that ignores the top five lines of output, compares the current ASCII text to an existing file, and writes the differences to another file:

```
| tail +5 | diff - file > file.diff
```

## Exiting from TotalView

You can exit from TotalView by selecting the **File > Exit** command. You can select this command in the Root, Process, and Variable Windows.

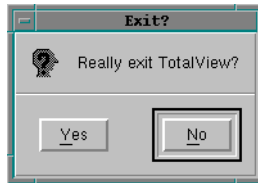


FIGURE 22: **File > Exit Dialog Box**

After you enter one of these commands, TotalView displays a dialog box. Select **Yes** to exit. Otherwise, select **No**. As TotalView exits, it kills all programs and processes that it started. However, programs and processes used that TotalView did not start, continue to execute.

**NOTE** If you have a CLI window open, TotalView also closes this window. Similarly, if you type “exit” within the CLI, the CLI will close TotalView windows.





# Setting Up a Debugging Session

This chapter explains how to set up basic TotalView sessions. It also describes some common commands and procedures. For information on setting up remote debugging sessions, see Chapter 4, *"Setting Up Remote Debugging Sessions"* on page 61. For information on setting up parallel debugging sessions, see Chapter 5, *"Setting Up Parallel Debugging Sessions"* on page 77.

In this chapter, you will learn about:

- Compiling Programs
- Starting TotalView
- Loading Executables
- Attaching to Processes
- Detaching from Processes
- Examining a Core File
- Processes and Thread State
- Handling Signals
- Setting Search Paths
- Setting Command Arguments
- Setting Input and Output Files
- Setting Preferences
- Setting Environment Variables
- Monitoring TotalView Sessions

## Compiling Programs

Before you start to debug a program, you must compile the program with the appropriate options and libraries for your situation. Table 4 presents

some general considerations, but you should check Appendix A, “*Compilers and Platforms*,” on page 311 to determine the exact syntax and any other considerations.

TABLE 4: Compiler Considerations

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually <code>-g</code> )	Generates debugging information in the symbol table.	Before debugging <i>any</i> program with TotalView.
Optimization option (usually <code>-O</code> )	Moves code to optimize execution of program.  Some compilers do not let you use the <code>-O</code> option with the <code>-g</code> option.  Even if you can, we recommend against it because using the <code>-O</code> option when debugging your program can produce strange results.	After you finish debugging your program with TotalView.
Multiprocess programming library (usually <code>dbfork</code> )	Uses special versions of the <code>fork()</code> and <code>execve()</code> system calls.  Using <code>dbfork</code> is discussed in “ <i>Linking with the dbfork Library</i> ” on page 317.	Before debugging a multiprocess program that explicitly calls <code>fork()</code> or <code>execve()</code> .  Refer to “ <i>Processes That Call fork()</i> ” on page 211 and “ <i>Processes That Call execve()</i> ” on page 211.

## Starting TotalView

TotalView can operate on programs that run in many different computing environments and which make use of a variety of parallel processing modes. This section looks at general ways to start TotalView. More detailed information can be found in *TotalView Command Syntax* on page 289.

The basic command structure for starting TotalView is:

```
totalview [ executable [ corefile ] ] [ options ]
```

where *executable* specifies the name of the executable file to be debugged and *corefile* specifies the name of the core file to be debugged.

**NOTE** If you are starting the CLI, you will type “totalviewcli” rather than “total-view”.

Here are some of the common ways to start TotalView:

**totalview** Starts TotalView without loading a program or core file. After TotalView starts, you can load a program by using the **File > New Program** command from the Root Window.

**totalview** *executable* Starts TotalView and loads the *executable* program.

**totalview** *executable corefile* Starts TotalView and loads the *executable* program and the *corefile* core file.

**totalview** *executable* **-a** *args* Starts TotalView and passes all following arguments (specified by *args*) to the *executable* program. If you use the **-a** option, it must appear after all other TotalView options on the command line. If you do not use **-a** and want to add arguments after TotalView loads your program, use the **Process > Startup** command.

**totalview** *executable* **-remote** *hostname\_or\_address[:portnumber]* Starts TotalView on the local host and the TotalView Debugger Server (**tvdsvr**) on the remote host. Loads the program specified by *executable* for remote debugging. You can specify a host name or a TCP/IP address, and optionally, a TCP/IP port number for *portnumber*.

Within a debugging session, you can reload programs and shared libraries. If these executable files have changed since you started debugging your program and you do not use the **Group > Reload Symbols** command (see “Reloading a Recomplied Executable” on page 38) to reload them, TotalView displays an error message because its symbol tables would not be up-to-date.

For more information on:

- Debugging parallel programs such as MPI, PVM, or HPF, refer to Chapter 5, “*Setting Up Parallel Debugging Sessions*” on page 77.
- The **totalview** command, refer to Chapter 13, “*TotalView Command Syntax*” on page 289.
- Remote debugging, refer to “*Starting the TotalView Debugger Server*” on page 61 and Chapter 14, “*TotalView Debugger Server (tvdsrv) Command Syntax*” on page 303.

## Loading Executables

TotalView can debug programs on local and remote hosts and programs accessed over serial lines. The **File > New Program** command located on either the Root or Process Windows loads local and remote programs, core files, and processes that are already running. Figure 23 shows the **File > New Program** dialog box.

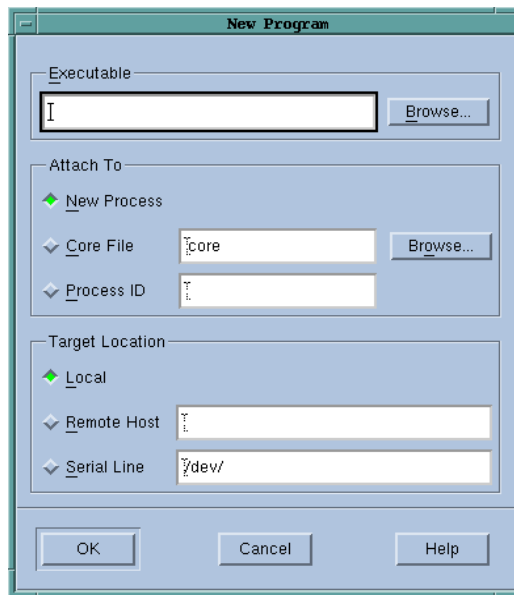


FIGURE 23: **File > New Program** Dialog Box

This dialog box lets you:

#### ■ Load a new executable

Type its path name into the **Executable** field

#### ■ Load a core file

Type its name into the **Core File** field *and* type the associated executable's path name into the **Executable** field.

#### ■ Load a process ID

Type its process ID into the **Process ID** field *and* type the associated executable's path name into the **Executable** field.

If you want to debug a program on a remote machine, enter the host name or IP address of the remote machine in the **Remote Host** field. (If the program is local, make sure that the **Local** button is selected.)

You can use a full or relative path name in the **Executable** and **Core File** fields. If you enter a simple file name, TotalView searches for it in the list of directories specified with the **File > Search Path** command or named in your **PATH** environment variable.

If you enter an executable name and **New Process** is selected, TotalView always loads a new copy of your program. This means that if it is already loaded, you will get another copy. In contrast, if you are trying to reload an already existing program, TotalView simply *pops* the program's Process Window. That is, it makes the window visible.

Debugging over a serial line is discussed in "Debugging Over a Serial Line" on page 72.

### More on Loading Remote Executables

If TotalView fails to automatically load a remote executable, you may need to disable the autolaunch feature for this connection and start the TotalView Debugger Server (**tvdsvr**) manually. Then, you can specify *hostname:portnumber* in step 2, where *portnumber* is the TCP/IP port number on which the debugger server is communicating with TotalView. Refer to "Starting the TotalView Debugger Server" on page 61 for more information.

**NOTE** You cannot examine core files on remote nodes.

You can connect to a remote machine in two ways: with the **–remote** option on the command line when you start TotalView or with the **File > New Program** command after you start TotalView.

You can also attach to a remote process by first connecting to a remote host using the **File > New Program** command and then displaying the Unattached Page of the Root Window. You can now attach to these processes by diving into them.

- 1 Connect to the remote host. For details, see “*Starting the TotalView Debugger Server*” on page 61.
- 2 After connecting to the remote host, bring up a list of unattached processes. You can attach to these processes by diving into them. For details, see “*Attaching Using the Unattached Page*” on page 39.

**NOTE** If TotalView supports a parallel process runtime library (for example, MPI, PVM, or HPF), it automatically connects to remote hosts. For more information, see Chapter 5, “*Setting Up Parallel Debugging Sessions*” on page 77.

For details on the syntax for the **–remote** command-line option, see “*Starting TotalView*” on page 34.

## Reloading a Recompiled Executable

If you edit and recompile your program while you are debugging it, you can load the updated program, as follows:

- 1 Confirm that all processes using the executable have exited. If they have not, invoke the **Process > Detach** or **Group > Delete** commands from the Process Window.
- 2 Confirm that duplicate copies of the process do not exist by using the shell’s **ps** command. If duplicate processes exist, delete them using the shell’s **kill** command.
- 3 In the Process Window, select the **Group > Reload Symbols** command. TotalView updates the Process Window with the new source file and loads the new executable file.

## Attaching to Processes

If a program you are testing is hung or looping (or misbehaving in some other way), you can attach to it while it is running. You can attach to single processes, multiprocess programs, and remote processes.

To attach to a process, either use the Unattached Page within the Root Window or use the **File > New Program** commands located on the Root and Process Windows. (Using the Unattached Page is easier if the process is listed. However, if it is not there, you must use the **File > New Program** command.)

You may also have to use the **New Program** dialog box to attach to a process when TotalView can determine the executable file name using the process listing in the Unattached Page.

If the process or any of its children calls the **execve()** routine, you may need to attach to it by creating a new Process Window. This is because TotalView uses the **ps** command on most platforms to obtain the name of the process executable. Since **ps** can give incorrect names, TotalView may not find it.

**NOTE** When you exit from TotalView, TotalView kills all programs and processes that it started. However, programs and processes that were executing before you brought them under TotalView's control continue to execute.

### Attaching Using the Unattached Page

To attach to a process using the Unattached Page, go to the Root Window and complete the following steps:

- 1 Select the Unattached Page Tab.

This page lists the process ID, status, and name of each process associated with your username. The processes that appear dimmed are those that are being debugged or those that TotalView will not allow you to debug (for example, the TotalView process itself).

The processes at the top of Figure 24 are all local. The remaining processes are remote.

If you are debugging a remote process, this page also shows processes running under your username on each remote host name. You can at-

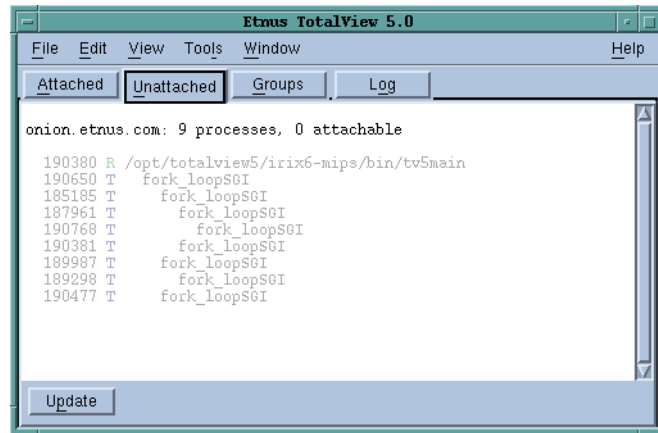


FIGURE 24: Unattached Page

attach to any of these remote processes. For more information on remote debugging, refer to "Starting the TotalView Debugger Server" on page 61 and Chapter 14, "TotalView Debugger Server (tvdsrv) Command Syntax" on page 303.

- 2 Dive into the process you wish to debug by double-clicking on it. A Process Window appears. The right arrow points to the current program counter (PC), indicating where the program was executing when TotalView attached to it.

## Attaching Using File > New Program

To attach to a process by using the Root Window's **File > New Program** command, follow these steps:

- 1 Use the **ps** shell command to obtain the process ID (PID) of the process.
- 2 Select the **File > New Program** command. TotalView displays the dialog box shown in Figure 25.

Enter a file name in the **Executable** field. This name can be a full or relative path name. If you supply a simple file name, TotalView searches for it in the directories specified with the **File > Search Path** command and specified by your **PATH** environment variable.

Enter the process ID (PID) of the *unattached* process into the **Process ID** field.



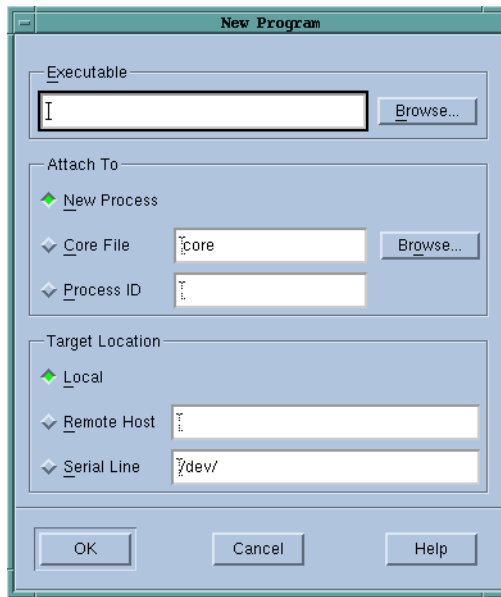


FIGURE 25: File &gt; New Program Dialog Box

### 3 Select OK.

If the executable is a multiprocess program, TotalView will ask if you want to attach to all relatives of the process. To examine all processes, select **Yes**.

If the process has children that call `execve()`, TotalView tries to determine each child's executable. If TotalView cannot determine the executable, you must delete (*kill*) the parent process and start it again using TotalView.

Finally, a Process Window appears. The right arrow points to the current program counter (PC), which is where the program was executing when TotalView attached to it.

## Detaching from Processes

You can detach from processes that TotalView did not create when you are done with them by using the following procedure:

- 1 Open a Process Window on the process.
- 2 If you want to send the process a signal, select the **Thread > Continuation Signal** command. Choose the signal that TotalView should send to the process when it detaches from the process. For example, to detach from a process and leave it stopped, set the continuation signal to **SIGSTOP**.

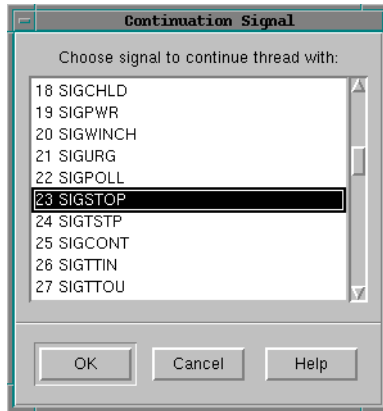


FIGURE 26: **Thread > Continuation Signal** Dialog Box

- 3 Select the **Process > Detach** command.

When you detach from a process, TotalView removes all breakpoints that were set within it.

## Examining a Core File

If a process encounters a serious error and dumps a core file, you can examine it using one of the following methods:

- Start TotalView as follows:  
`totalview filename corefile [ options ]`
- Select the **File > New Program** command from the Root Window. In the middle section of the dialog box, type the name of the core file in the **Core File** field, and then select **OK**.

**NOTE** You can only debug local core files. You can, however, debug core files at a remote location if you log on to the remote machine and then start TotalView upon the now local core file. In this case, TotalView is running on the remote machine (that is, TotalView is now *local* to the machine upon which the application and core file reside).

The Process Window displays the core file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when it dumped core. The title bar of the Process Window names the signal that caused the core dump. The right arrow in the tag field of the Source Pane indicates the value of the program counter (PC) when the process encountered the error.

You can examine the state of all variables at the time the error occurred. “*Examining and Changing Data*” on page 153 contains more information.

If you start a process while you are examining a core file, TotalView stops using the core file and starts a fresh process using that executable.

## Processes and Thread State

Process and thread state is displayed in:

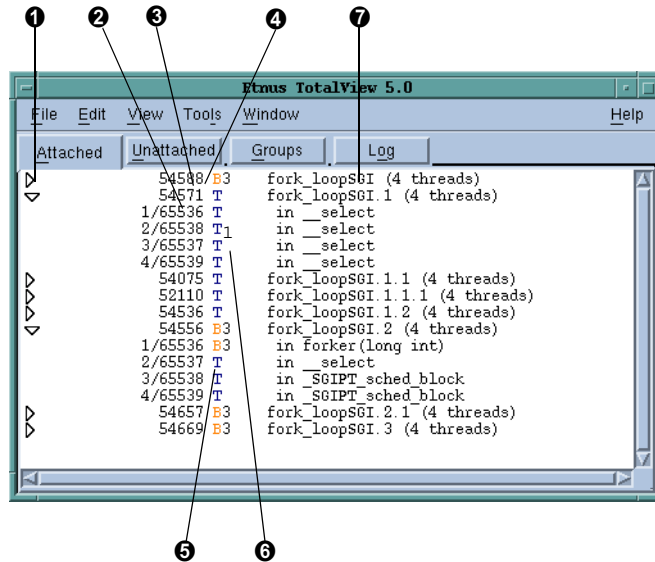
- The Attached Page of the Root Window, for processes and threads.
- The Unattached Page of the Root Window, for processes.
- The process and thread status bars of the Process Window.
- The Threads Pane of the Process Window, for threads.

Figure 27 shows TotalView displaying process state information in the Attached Pane:

The status of a process includes the process location, the process ID, and the state of the process. (These characters are explained in “*Attached Process States*” on page 44.)

The Unattached Page lists all processes associated with your username. The information in this window is similar to the information in the Attach Page, differing only in that processes being debugged are dimmed out.

**NOTE** If, as they are on some systems, the TotalView-assigned thread ID and the system-assigned thread ID are the same, TotalView displays only one ID value.



- ❶ Collapse/expand toggle
- ❷ TotalView thread ID (TID)
- ❸ System thread ID (SYSTID)
- ❹ Process ID (PID)
- ❺ Process status
- ❻ Action point ID number
- ❼ Program name

FIGURE 27: Attached Page Showing Process and Thread Status

The status bars in the Process Window display similar information. Here's an example:

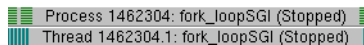


FIGURE 28: Process and Thread Labels in the Process Window

## Attached Process States

Process and thread state is displayed using the following symbols:

TABLE 5: Attached Process and Thread States

State Code	State Name
blank	Exited or never created
B	At breakpoint

TABLE 5: Attached Process and Thread States (Continued)

State Code	State Name
E	Error <i>reason</i>
K	In kernel
M	Mixed
R	Running
T	Stopped <i>reason</i>
W	At watchpoint

The **Error** state usually indicates that your program received a fatal signal from the operating system. Signals such as **SIGSEGV**, **SIGBUS**, and **SIGFPE** can indicate an error in your program. See “*Handling Signals*” on page 45 for information on controlling how TotalView handles signals that your program receives.

## Unattached Process States

The state information for a process displayed in the Unattached Page is derived from the system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the system.

Table 6 summarizes the possible states in the Unattached Page.

TABLE 6: Summary of Unattached Process States

State Code	State
I	Idle
R	Running
S	Sleeping
T	Stopped
Z	Zombie

## Handling Signals

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. You can do this by using:

- A dialog box (described in this section)
- An X resource (see `totalview*signalHandlingMode` on page 281)
- The `-signalHandlingMode` command-line option to the `totalview` command (refer to “*TotalView Command Syntax*” on page 289)

Unless you tell it otherwise, here is how TotalView handles UNIX signals:

TABLE 7: Default Signal Handling Behavior

Signals that TotalView Passes Back to Your Program		Signals that TotalView Treats as an Error	
SIGHUP	SIGIO	SIGILL	SIGPIPE
SIGINT	SIGIO	SIGTRAP	SIGTERM
SIGQUIT	SIGPROF	SIGIOT	SIGTSTP
SIGKILL	SIGWINCH	SIGEMT	SIGTTIN
SIGALRM	SIGLOST	SIGFPE	SIGTTOU
SIGURG	SIGUSR1	SIGBUS	SIGXCPU
SIGCONT	SIGUSR2	SIGSEGV	SIGXFSZ
SIGCHLD		SIGSYS	

TotalView uses the **SIGTRAP** and **SIGSTOP** signals internally. If the process encounters either signal, TotalView neither stops the process with an error nor passes the signal back to your program. Further, you cannot alter the way TotalView uses these signals.

On some systems, hardware registers can affect how signals such as **SIGFPE** are handled. For more information, refer to “*Interpreting Status and Control Registers*” on page 151 and Appendix C, “*Architectures*,” on page 337.

**NOTE** On SGI machines, setting the `TRAP_FPE` environment variable to any value indicates that your program will trap underflow errors. If you set this variable, however, you will also need to use the **File > Signals** dialog box to indicate what TotalView should do with **SIGFPE** errors. (In most cases, you will set **SIGFPE** to *Resend*.) As an alternative, you can use the `-signal_handling_mode` “*action\_list*” option (see page 299) or the `totalview*signalHandlingMode` X resource (see page 281) to “*Resend=SIGFPE*”.

You can change the signal handling mode by going to the Process Window and selecting the **File > Signals** command. The dialog box shown in Figure 29 appears.

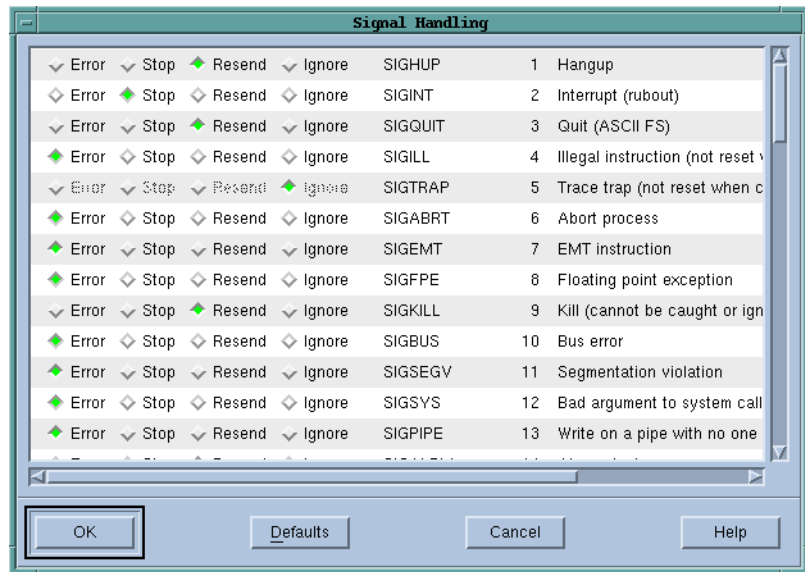


FIGURE 29: File &gt; Signals Dialog Box

**NOTE** The signal names and numbers shown in the dialog box are platform-specific.

When your program encounters an error signal, TotalView stops all related processes. If you do not want this behavior, deselect the **Stop control group on error signal** (which is found on the **Options** Page of the **File > Preference's** dialog box).

Also by default, when your program encounters an error signal, TotalView opens or raises the Process Window. Deselecting the **Open process window on error signal** check box, found on the **Options** Page of the **File > Preference's** dialog box, tells TotalView that it should not open or raise the window. You can also use an X resource (`totalview*popOnError` on page 281) or the `-pop_on_error` command-line option to create a default setting for this check box.

If processes in a multiprocess program encounter an error, TotalView only opens a Process Window for the *first* process that encounters an error. This stops TotalView from filling up the screen with Process Windows.

If you select the **Open process window at breakpoint** check box, which is found on the **File > Preference's Action Points** Page, TotalView opens or raises the Process Window when your program reaches a breakpoint. You can also set TotalView's default behavior using the **-pop\_at\_breakpoint** command-line option or an X Resource (**totalview\*popAtBreakpoint** on page 280)

If necessary, scroll the signal list to the signal being changed. Make your changes by selecting one of the following radio buttons:

<b>Error</b>	Stops the process, places it in the error state, and displays an error in the title bar of the Process Window. If the <b>Stop control group on error signal</b> check box is selected, TotalView also stops all related processes. You should select this signal handling mode for severe error conditions such as <b>SIGSEGV</b> and <b>SIGBUS</b> signals.
<b>Stop</b>	Stops the process and places it in the stopped state. Select this signal handling mode if you want TotalView to handle this signal the same as a <b>SIGSTOP</b> signal.
<b>Resend</b>	Sends the signal to the process. If your program contains a signal handling routine, you should use this mode for all the signals that it handles. By default, the common signals for terminating a process ( <b>SIGKILL</b> and <b>SIGHUP</b> ) use this mode.
<b>Ignore</b>	Discards the signal and restarts the process without a signal.

**NOTE** Do not use Ignore mode for fatal signals, such as **SIGSEGV** and **SIGBUS**. If you do, TotalView can get caught in a signal/resignal loop with your program; the signal will immediately recur because the failing instruction will reexecute repeatedly.

## Setting Search Paths

If your source code, executable, or object files reside in different directories, set search paths for these directories with the **File > Search Path** command. TotalView searches the following directories (in order):



- 1 The current working directory (.).
- 2 The directories you specify by using the **File > Search Path** command in the exact order you enter them in the dialog box. When you enter them, enter one path name to a line.
- 3 If you specified a full path name for the executable when you started TotalView, TotalView searches this directory.
- 4 The directories specified in your **PATH** environment variable.

These search paths apply to *all* processes that you are debugging. After TotalView responds to your selection of the **File > Search Path** command by displaying the dialog box, shown in Figure 30, you can enter paths into the text control

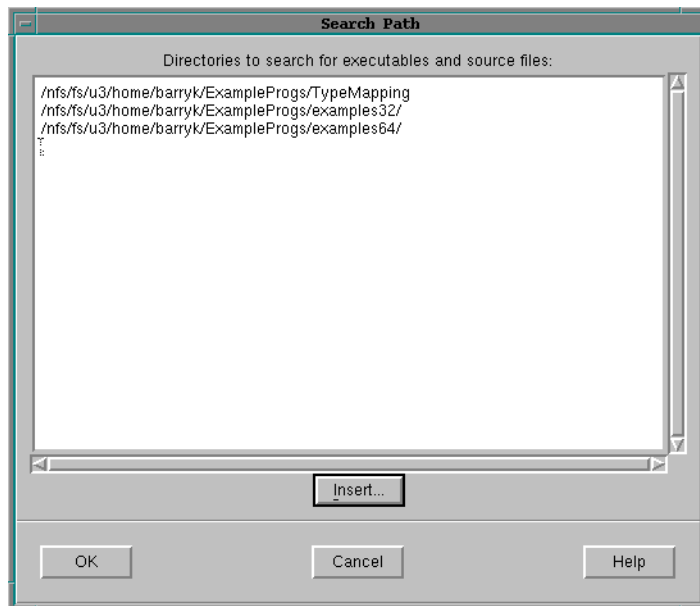


FIGURE 30: **File > Search Path** Dialog Box

You must enter directories in the order you want them searched. In addition, you must enter each directory on its own line.

- You can type path names directly.
- You can cut and paste directory information.

- You can use the **insert** button to tell TotalView to display a file browser dialog box that lets you browse through the file system, interactively selecting directories. The dialog box is shown in Figure 31.



FIGURE 31: **Add Directories Dialog Box**

The current working directory (.) within the **File > Search Path** dialog box is the first directory listed in the window. Relative path names are interpreted as being *relative* to the current working directory.

Note that if you remove the current working directory, TotalView reinserts it at the top of the directory.

After you change this list of directories, TotalView again searches for the source file that is currently displayed in the Process Window.

You can also specify search directories using an X Window System resource. Refer to **totalview\*searchPath** on page 281.

## Setting Command Arguments

When TotalView creates a process, it uses the name of the file containing the executable code for the process's program name. If your program re-

quires command-line arguments, you must set these arguments *before* you start the process, as follows:

- 1 Select the Arguments Tab within the **Process > Startup Parameters** dialog box. Here is the Arguments Page.

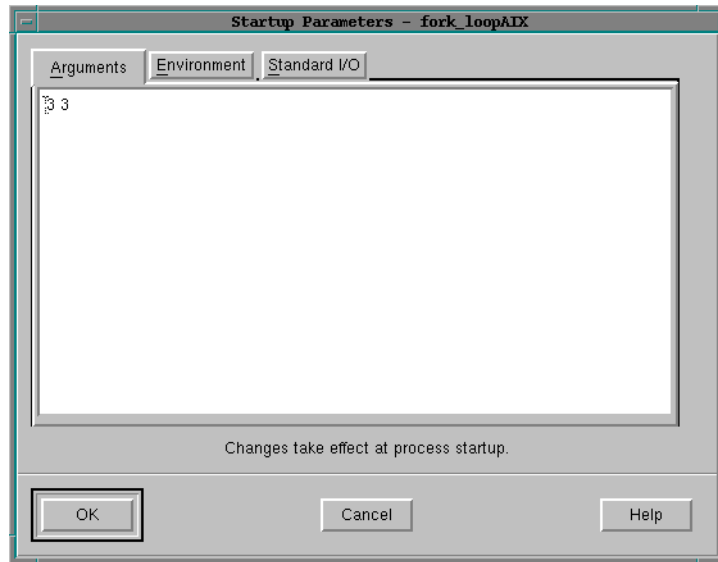


FIGURE 32: **Process > Startup Parameters Dialog Box: Arguments Page**

- 2 Type the arguments to be passed to the program. Separate each argument with a space, or place each argument on a separate line. If an argument has spaces in it, enclose the entire argument in double quotes. When you are done, select **OK**.

You can also set command-line arguments with the `-a` option of the **totalview** command, as discussed in "Starting TotalView" on page 34.

## Setting Input and Output Files

Before TotalView begins executing a program, it determines how it will handle standard input (**stdin**) and standard output (**stdout**). Unless you tell it otherwise, **stdin** and **stdout** use the shell window from which TotalView was invoked.

You can redirect **stdin** or **stdout** to a file by completing these steps from the Process Window before you start executing your program:

- 1 Select the Standard I/O Tab from the dialog box displayed when you invoke the **Process > Startup Parameters** command. This page is shown in Figure 33.

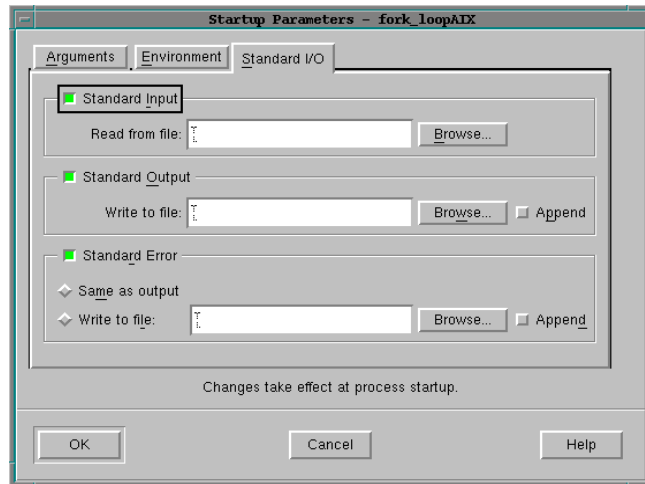


FIGURE 33: **Process > Startup Parameters** Dialog Box: Standard I/O Page

- 2 Type the name of the file, relative to your current working directory. Entering names in these boxes produces the same effect as if you had used a `<`, `>`, or `>&` symbols while in the shell.
- 3 Select **OK**.

If the **Append** check box is set, TotalView opens this file and appends new information to the end of the file.

If the **Same as output** check box is selected, TotalView writes **stderr** information to the same output file as **stdout**.

## Setting Preferences

Using the **File > Preferences** command, you can tailor what TotalView does in many situations. This section contains an overview of the many preferences you can set. Complete information can be found in the Help.

Some settings such as the prefixes and suffixes looked at when loading dynamic libraries can be different from operating system to operating system. Consequently, if the setting can differ, TotalView automatically makes the setting unique for your operating system and this is done transparently.

As TotalView stores a unique version for each platform, you will not see a preference set on one platform when you are executing on another. In general, this applies to the server launch strings and dynamic library paths.

You will find information on setting attributes using X resources, setting and overriding preferences, and using options in the next section.

- **Options.** This page contains radio buttons that are either general in nature or that influence different parts of the system.

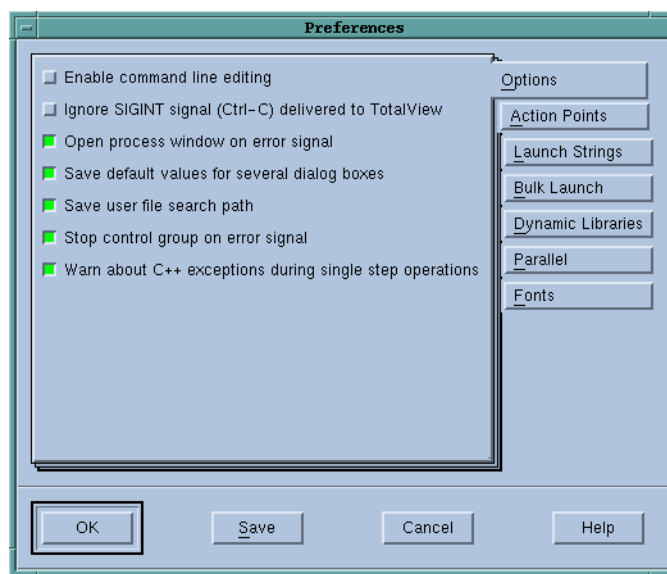


FIGURE 34: File > Preferences Dialog Box: Options Page

- **Action Points.** The commands on this page indicate what else is stopped, if anything, when TotalView encounters an action point, the scope of the action point, automatic saving and loading, and if TotalView should open a Process Window for the process encountering a breakpoint.

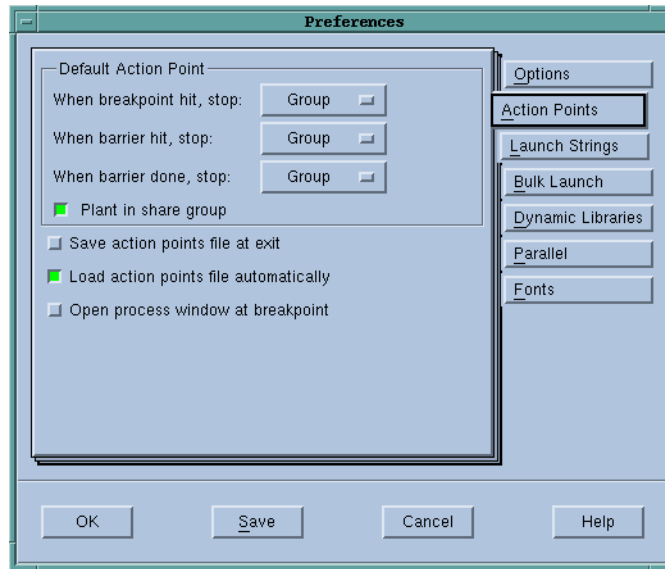


FIGURE 35: **File > Preferences Dialog Box: Action Points Page**

- **Launch Strings.** See Figure 36 on page 55. The three areas of this page let you set the launch string used when TotalView launches its remote debugging server, the Visualizer, and a source code editor. Notice that there are default values for these launch strings.
- **Bulk Launch.** See Figure 37 on page 55. The fields and commands on this page configure TotalView's bulk launch. See Chapter 4 for more information.
- **Dynamic Libraries.** See Figure 38 on page 56. This page lets you control which symbols are added to TotalView when it loads a dynamic library.
- **Parallel.** See Figure 39 on page 56. This page lets you define what will occur when your program goes parallel.
- **Fonts.** See Figure 40 on page 57. Use this page to specify the fonts used in the user interface and when TotalView displays your code.

## Setting Preferences, Options, and X Resources

While preferences are the best way to set many of TotalView's features and characteristics, TotalView also lets you set features and characteristics using X resources and command-line options.

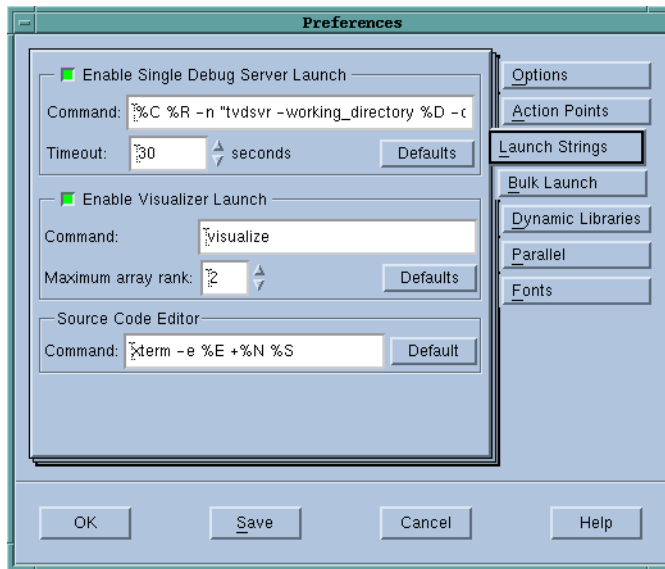


FIGURE 36: File &gt; Preferences Dialog Box: Launch Strings Page

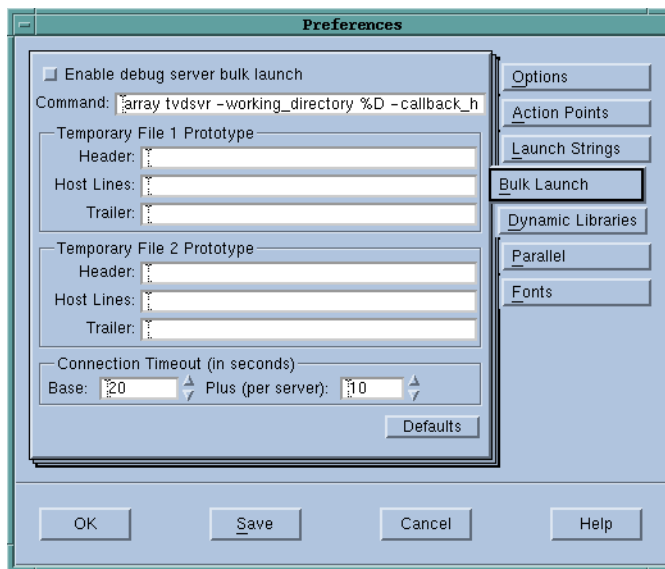


FIGURE 37: File &gt; Preferences Dialog Box: Bulk Launch Page

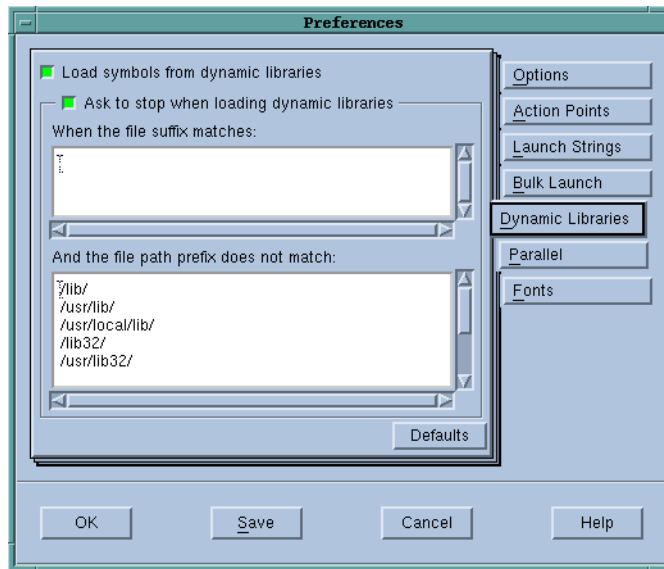


FIGURE 38: File &gt; Preferences Dialog Box: Dynamic Libraries Page

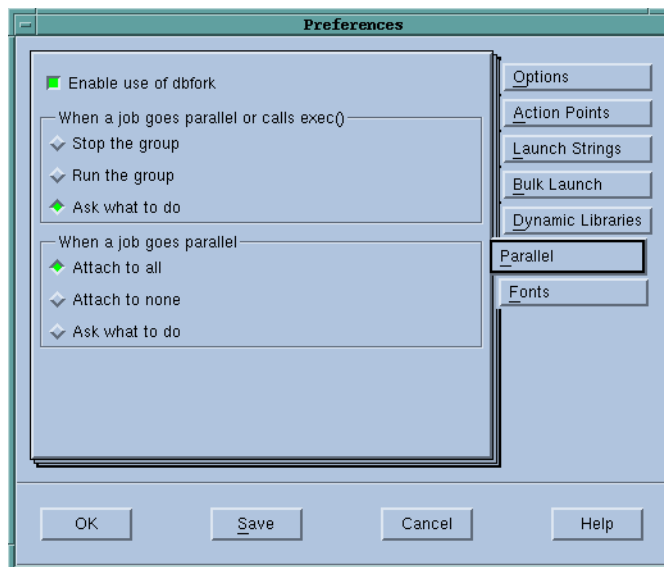


FIGURE 39: File &gt; Preferences Dialog Box: Parallel Page



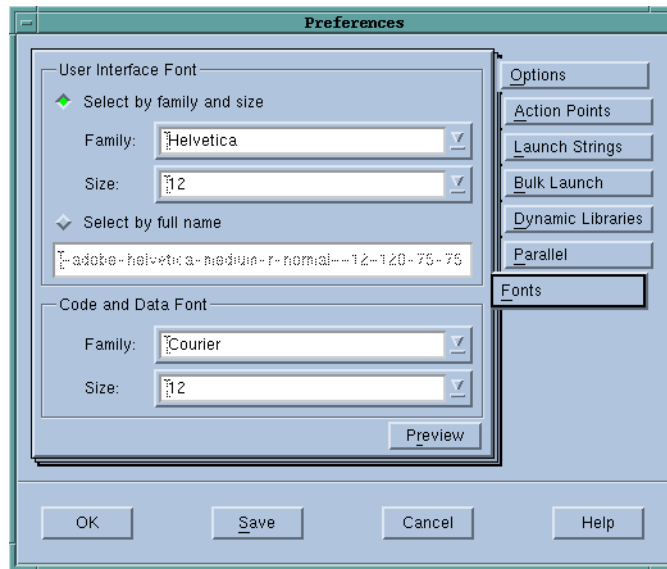


FIGURE 40: File &gt; Preferences Dialog Box: Fonts Page

Older versions of TotalView did not have a preference system. Instead, you needed to set values in your `.Xdefaults` file. For example, setting `totalview*autoLoadBreakpoints` to true tells TotalView that it should automatically load a breakpoint file when it loads an executable. Because this option can also be set as a preference, this resource entry has been *deprecated*. That is, it still can be used but future releases may not support it. You can now set this variables using the CLI's `dset` command.

Some resources are not yet part of TotalView's preferences. The action or state set by the resource must continue to be set using an `.Xdefaults` file.

Chapter 12, "X Resources" on page 275 describes the resources you can set. Deprecated resources are not included in this appendix.

Preferences and X resources indicate states and characteristics that you intend to exist in all of your TotalView sessions. In some cases, you may want to set a state for one session or you may want to override one of your global settings. This is the function of the command-line options described in Chapter 13, "TotalView Command Syntax" on page 289.

For example, you can use the `-fixed_font_size` to override the font size indicated in the preference in the current TotalView session. Any changes you make use an option are not remembered in other sessions.

## Setting Environment Variables

You can set and edit the environment variables that TotalView passes to processes. When TotalView creates a new process, it passes a list of environment variables to the process.

If the Environment Page within the **Process > Startup Parameters** dialog box is empty, new processes inherit its environment variables from TotalView or `tvdsrv`.

**NOTE** If you add environment variables, the process no longer inherits environment variables; it only receives the variables that you enter in this dialog box. Therefore, if you want to add additional variables to those inherited that would be inherited, you must enter the variables being inherited in addition to the ones you are adding.

An environment variable is specified as *name=value*. For example, the following definition creates an environment variable named **DISPLAY** whose value is `unix:0.0`:

**DISPLAY=unix:0.0**

To add, delete, or modify environment variables, select the Environments Tab from the dialog box displayed when you invoke the **Process > Startup Parameters** command. See Figure 41.

In the displayed dialog box, place each environment variable on a separate line.

The actions you can now perform are:

- To change the name or value of an environment variable, edit the line.
- To add a new environment variable, insert a new line and specify the name and value.
- To delete an environment variable, delete the line. If you delete all the lines, the process inherits TotalView or `tvdsrv`'s environment.

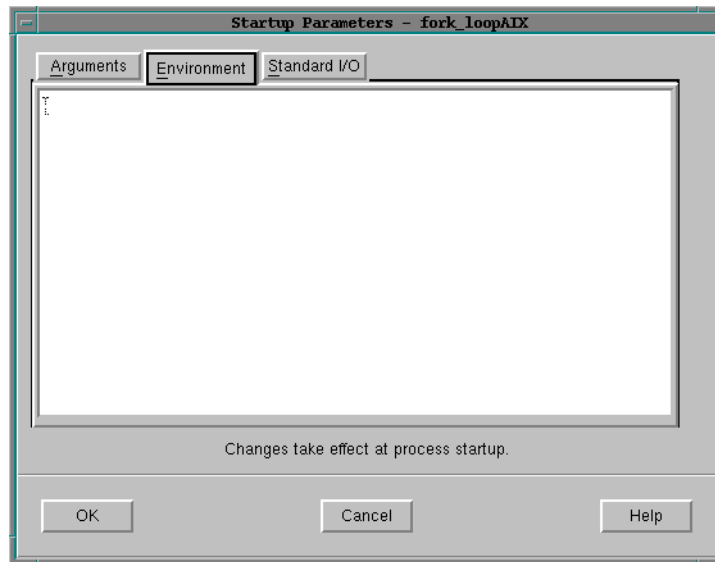


FIGURE 41: **Process > Startup Parameters Dialog Box: Environment Page**

## Monitoring TotalView Sessions

TotalView logs all significant events occurring for all processes being debugged. To view the event log, select the Root Window's Log Tab. This page displays a sequential list of these events. See Figure 42 for an example.

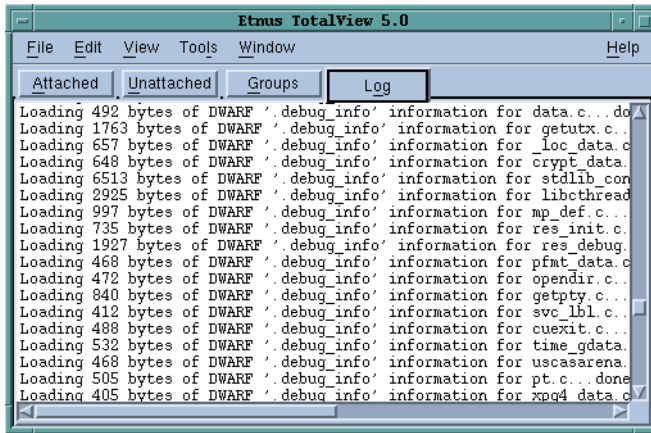


FIGURE 42: Root Window Log Page

# Setting Up Remote Debugging Sessions

This chapter explains how to set up TotalView remote debugging sessions. This chapter discusses:

- Starting the TotalView Debugger Server
- Debugging Over a Serial Line

## Starting the TotalView Debugger Server

Debugging a remote process with TotalView only differs from debugging a native process in that:

- TotalView works with other TotalView processes running on remote machines. This remote TotalView process is called the TotalView Debugger Server (**tvdsvr**).
- The performance of your session depends on the performance of the network between the native and remote machines. If the network is overloaded, debugging can be slow.

Unless you tell it otherwise, TotalView automatically launches **tvdsvr** in one of the following ways.

- It can launch a **tvdsvr** on each remote host independently. This is called *single process server launch*.
- It can launch all remote processes at the same time. This is called *bulk server launch*.

Autolaunching greatly simplifies the debugging remote processes since you do not need to take any action to debug remote processes.

## Single Process Server Launch Options

The **Remote Debug Server Launch** preferences within the **Launch Strings** Page of the **File > Preferences** dialog box lets you change the command used to launch remote servers, disable autolaunch, and alter the connection timeout used by TotalView when it launches **tvdsvr**.

Here is the **Launch Strings** Page:

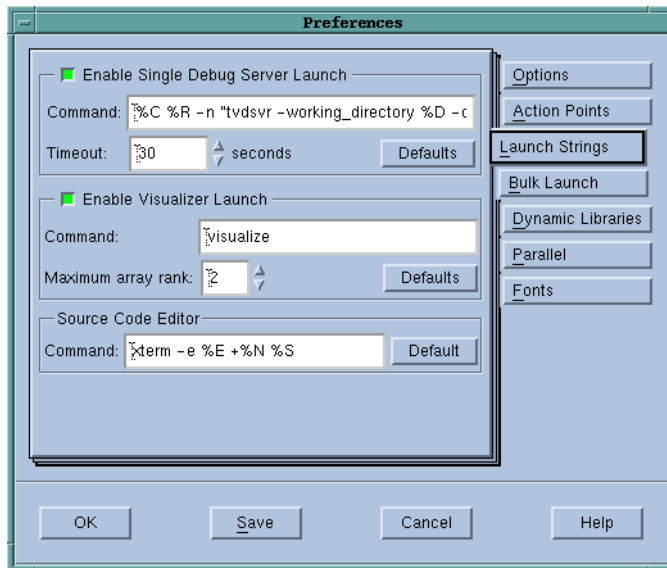


FIGURE 43: **File > Preferences: Server Launch Strings Page**

### Enable Single Debug Server Launch

If this check box is selected, TotalView will autolaunch the TotalView Debugger Server (**tvdsvr**).

### Command

If **Enable Single Debug Server Launch** is selected, TotalView will use this command to launch **tvdsvr**. For information on this command and its options, see “*Single Process Server Launch Command*” on page 66.

### Timeout (Sec.)

After TotalView automatically launches **tvdsvr**, it waits 30 seconds for **tvdsvr** to respond with a successful connection message. If the connection is not made in

this time, TotalView times out. You can change this by entering a value from 1 to 3600 seconds (1 hour).

In addition, you can preset the timeout value using a TotalView preference. See the online help for more information.

If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started the debugger) before the timeout expires, select **Yes** in the **Question** dialog box.

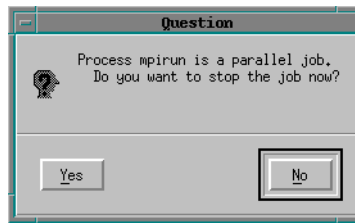


FIGURE 44: Stop Job Question Dialog Box

### Defaults

If you make a mistake or decide you want to go back to TotalView's default settings, select this button.

This command also overrides changes you made using an X resource. TotalView does not immediately change settings after you press the **Defaults** button; instead, it waits until you select the **OK** button.

## Bulk Launch Window Options

The fields within the **File > Preferences's Bulk Launch** Page lets you change the bulk launch command, disable bulk launch, and alter connection timeouts used by TotalView when it launches **tvdsvr** programs.

Figure 45 shows this page.

### Enable debug server bulk launch

If the check box is selected, TotalView will bulk launch the TotalView Debugger Server (**tvdsvr**). By default, bulk launch is disabled.

### Command

If bulk launch is enabled, TotalView will use this command to launch **tvdsvr**. For information on this com-

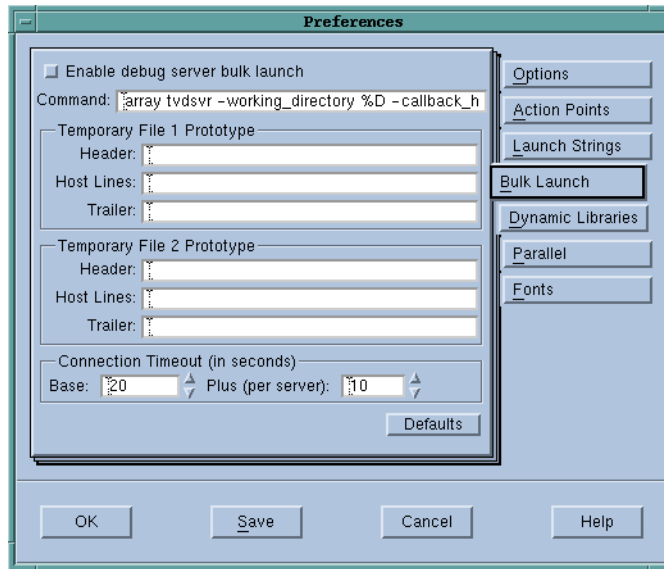


FIGURE 45: File &gt; Preferences Bulk Launch Page

mand and its options, see “Bulk Server Launch on an SGI MIPS Machine” on page 68 and “Bulk Server Launch on an IBM RS/6000 AIX Machine” on page 69.

### Temporary File Prototypes

These six fields let you create temporary files in bulk launch operations. For information on these fields, see Chapter 14 “TotalView Debugger Server (tvdsrv) Command Syntax” on page 303.

### Connection Timeout (in seconds)

After TotalView launches **tvdsrv** processes, it waits 20 seconds (the **Base** time) plus 10 seconds for each server that it will launch for responses from successfully connected processes. (The text boxes let you change these values.) If connections are not made in this time, TotalView times out.

The **Base** timeout value can be from 1 to 3600 seconds (1 hour). The incremental **Plus** value is from 1 to 360 seconds. You can preset these values using TotalView preferences. See the online help for more information.



If you notice that TotalView fails to launch **tvdsvr** (as shown in the **xterm** window from which you started the debugger) before the timeout expires, select **Yes** in the Question dialog box that will appear.

#### Defaults

If you make a mistake or decide you want to go back to TotalView's default settings, select this button.

## Starting the Debugger Server Manually

If you cannot make TotalView's autolaunch feature work on your system, you can start **tvdsvr** manually. Unfortunately, this method is not completely secure: other users could connect to your instance of **tvdsvr** and begin using your UNIX UID.

Here is how you manually start **tvdsvr**:

- 1 Select the **Bulk Launch** Tab within the **File > Preferences** dialog box. (You can select this command from the Root Window or the Process Window) The dialog box shown in Figure 43 on page 62 appears.
- 2 Deselect the **Enable debug server bulk launch** check box within the **Bulk Launch** Tab of the **File > Preferences** dialog box to disable the autolaunch feature and then select **OK**.
- 3 Log in to the remote machine and start **tvdsvr**:

#### **tvdsvr -server**

If you do not (or cannot) use the default port number (4142), you will need to use the **-port** or **-search\_port** options. For details, refer to Chapter 14 "TotalView Debugger Server (*tvdsvr*) Command Syntax" on page 303.

After printing out the port number and the assigned password, the server begins listening for connections. Be sure to make note of the password; you will need to enter it later in step 5.

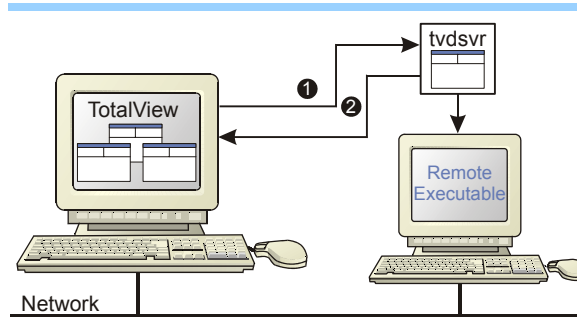
**NOTE** Because using the **-server** option is not completely secure, it must be explicitly enabled. (This is usually done by your system administrator.) For details, see "**-server**" on page 306.

- 4 From the Root Window, select the **File > New Program** command. Type the name in the **Executable** field and the *hostname:portnumber* in the **Remote Host** field.  
Select **OK**.

### 5 TotalView now tries to connect to **tvdsvr**.

When TotalView prompts you for the password, enter the password that **tvdsvr** displayed in step 3.

Figure 46 summarizes the steps used when you start **tvdsvr** manually.



- ❶ Makes connection
- ❷ Listens

FIGURE 46: Manual Launching of Debugger Server

## Single Process Server Launch Command

By default, TotalView uses the following command string when it automatically launches the debugger server for a single process:

```
%C %R -n "tvdsvr -working_directory %D -callback %L \
-set_pw %P -verbosity %V"
```

where:

- %C** Expands to the name of the server launch command being used. On most platforms, this is **rsh**. On HP machines, this command is **remsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, TotalView will use its value instead of its platform-specific default value.
- %R** Expands to the host name of the remote machine that you specified in the **File > New Program** command.
- n** Tells the remote shell to read standard input from **/dev/null**; that is, it immediately received an EOF signal.

**-working\_directory %D**

Makes **%D** the directory to which TotalView will be connected. **%D** expands to the absolute path name of the directory.

Using this option assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on the host and target machines.

After changing to this directory, the shell will invoke the **tvdsvr** command.

You must make sure that TotalView directory is on your path on the remote machine.

**-callback %L**

Establishes a connection from **tvdsvr** to TotalView. **%L** expands to the host name and TCP/IP port number (*hostname:port*) upon which TotalView is listening for connections from **tvdsvr**.

**-set\_pw %P**

Sets a 64-bit password. TotalView must supply this password when **tvdsvr** establishes a connection with it. **%P** expands to the password that TotalView automatically generated. For more information on this password, see Chapter 14 "TotalView Debugger Server (*tvdsvr*) Command Syntax" on page 303.

**-verbosity %V**

Sets the verbosity level of the TotalView Debugger Server. **%V** expands to the current TotalView verbosity setting.

You can also use the **%H** option with this command. This option is discussed in "Bulk Server Launch on an SGI MIPS Machine" on page 68.

To set the server launch command that will be invoked whenever you start TotalView, you can set a TotalView preference. See the online help for more information.

For information on the complete syntax of the **tvdsvr** command, refer to "TotalView Debugger Server (*tvdsvr*) Command Syntax" on page 303.

## Bulk Server Launch on an SGI MIPS Machine

On an SGI machine, the launch string used for a bulk server launch is similar to the single process server launch and is:

```
array tvdsvr -working_directory %D -callback_host %H \
             -callback_ports %L -set_pws %P -verbosity %V
```

where:

### **-working\_directory %D**

Makes **%D** the directory to which TotalView will be connected. **%D** expands to the absolute path name of the directory.

Note that the command assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on both the host and target machines.

After performing this operation, the TotalView Debugger Server is started.

### **-callback\_host %H**

Names the host upon which the callback is made. **%H** expands to the host name of the machine upon which TotalView is running.

### **-callback\_ports %L**

Names the ports on the host machines that are used for callbacks. **%L** expands to a comma-separated list of the host names and TCP/IP port numbers (*host-name:port,hostname:port...*) on which TotalView is listening for connections from **tvdsvr**.

### **-set\_pws %P**

Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. **%P** expands to a comma-separated list of 64-bit passwords that TotalView automatically generates. For more information, see Chapter 14 "TotalView Debugger Server (**tvdsvr**) Command Syntax" on page 303.

### **-verbosity %V**

Sets the verbosity level of the TotalView Debugger Server. **%V** expands to the current TotalView verbosity setting.

You must enable **tvdsrv**'s use of the array command by adding the following information to the `/usr/lib/array/arrayd.conf` file:

```
#
# Command that allow invocation of the TotalView Debugger
# server when performing a Bulk Server Launch.
#
command tvdsrv
    invoke /opt/totalview/bin/tvdsrv %ALLARGS
    user %USER
    group %GROUP
    project %PROJECT
```

For information on the complete syntax of the **tvdsrv** command, refer to Chapter 14 “*TotalView Debugger Server (tvdsrv) Command Syntax*” on page 303.

## Bulk Server Launch on an IBM RS/6000 AIX Machine

On an IBM RS/6000 AIX machine, the launch string used for a bulk server launch is:

```
%C %H -n "poe -pgmmodel mpmc -resd no -tasks_per_node 1
        -procs %N -hostfile %t1 -cmdfile %t2"
```

where the elements unique to TotalView are:

<b>%N</b>	The number of servers that will be launched.
<b>%t1</b>	A temporary file created by TotalView that contains a list of the hosts upon which <b>tvdsrv</b> will run.  TotalView generates this information by expanding the <b>%R</b> symbol entered within the <b>Bulk Launch</b> preferences.
<b>%t2</b>	A file that contains the commands to start the <b>tvdsrv</b> processes on each machine. TotalView creates these lines by expanding the following template:  <pre>tvdsrv -working_directory %D \       -callback %L -set_pw %P -verbosity %V</pre>

## Disabling Autolaunch

If after changing the autolaunch options, TotalView still cannot automatically start **tvdsrv**, you must disable the autolaunch and start **tvdsrv** manually. Here are three ways for doing this:

- Deselect the **Enable Single Debug Server Launch** check box in the **Launch Strings** Page of the **File > Preferences** dialog box.
- When you debug the remote process, as described in "Starting the TotalView Debugger Server" on page 61, enter a host name and port number in the bottom section of the **File > New Program** dialog box. This disables autolaunch for the current connection.
- Set a preference that disables autolaunch. For more information, refer to the online help. Note that setting this preference disables autolaunch for all TotalView sessions.

**NOTE** If you disable the autolaunch feature, you must start **tvdsrv** *before* you load a remote executable or attach to a remote process.

## Changing the Remote Shell Command

Some environments require that you create your own autolaunch command. You might do this, for example, if your remote shell command does not provide the security required by your site and you need to invoke remote processes by using a more secure command.

If you create your own autolaunch command, you must invoke **tvdsrv** by using the **-callback** and **-set\_pw** arguments.

If you are not sure whether **rsh** (or **remsh** on HP machines) works at your site, try typing "**rsh hostname**" (or "**remsh hostname**") from an **xterm**, where *hostname* is the name of the host upon which you want to invoke the remote process. If this command prompts you for a password, you must add the host name of the host machine to your **.rhosts** file on the target machine.

For example, you could use a combination of the **echo** and **telnet** commands:

```
echo %D %L %P %V; telnet %R
```

Once **telnet** establishes a connection to the remote host, you could use the **cd** and **tvdsrv** commands directly, using the values of **%D**, **%L**, **%P**, and **%V** that were displayed by the **echo** command. For example:

```
cd directory
tvdsrv -callback hostname:portnumber -set_pw password
```

If your machine does not have a command for invoking a remote process, you cannot use the autolaunch feature and should disable it.

For information on the **rsh** and **remsh** commands, refer to the manual page supplied with your operating system.

## Changing the Arguments

You can also change the command-line arguments passed to **rsh** (or whatever command you use to invoke the remote process).

For example, if the host machine does not mount the same file systems as your target machine, the debugger server may need to use a different path to access the executable being debugged. If this is the case, you could change **%D** to the directory used on the target machine.

If the remote executable reads from standard input, you cannot use the **-n** option with your remote shell command because this option causes the remote executable to receive an EOF immediately on standard input. If you omit **-n**, the remote executable reads standard input from the **xterm** in which you started TotalView. This means that you should invoke **tvdsrv** from another **xterm** window if your remote program reads from standard input. Here's an example:

```
%C %R "-working_directory %D -display hostname:0 -e tvdsrv \
    -callback %L -set_pw %P -verbosity %V"
```

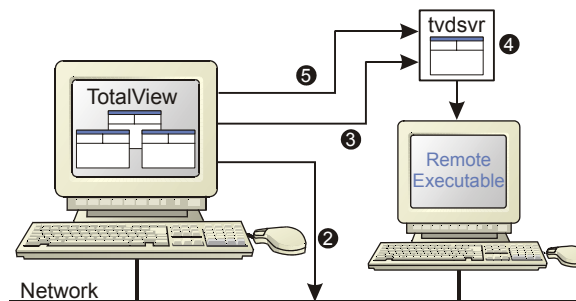
Now, each time TotalView launches **tvdsrv**, a new **xterm** appears on your screen to handle standard input and output for the remote program.

## Autolaunch Sequence

If you want to know more about autolaunch, here is the sequence of actions carried out by you, TotalView, and **tvdsrv**:

- 1 With the **File > New Program** command, you specify the host name of the machine on which you want to debug a remote process, as described in “*Starting the TotalView Debugger Server*” on page 61.
- 2 TotalView begins listening for incoming connections.
- 3 TotalView launches the **tvdsrv** process with the server launch command. (“*Single Process Server Launch Command*” on page 66 describes this command.)
- 4 The **tvdsrv** process starts on the remote machine.
- 5 The **tvdsrv** process establishes a connection with TotalView.

Figure 47 summarizes these actions.



- 2 Listens
- 3 Invokes commands
- 4 tvdsrv starts
- 5 Makes connection

FIGURE 47: Root Window Showing Process and Thread Status

## Debugging Over a Serial Line

TotalView allows you to debug programs over a serial line as well as TCP/IP sockets. However, if a network connection exists, you will probably want to use it because performance will be much better.

You will need to have two connections to the target machine. One connection is for the console and the other is for TotalView’s use. Do not try to use one serial line. TotalView cannot share a serial line with the console.



Figure 48 illustrates a TotalView debugging session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line, allowing you to type commands on the target host.

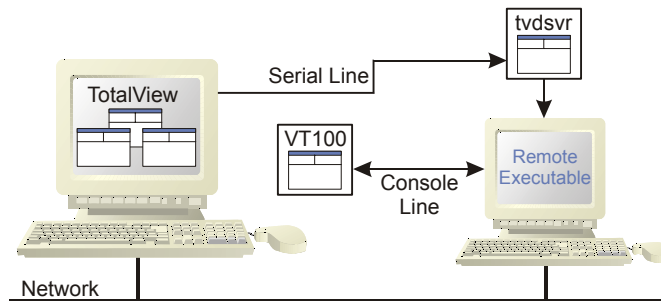


FIGURE 48: TotalView Debugging Session Over a Serial Line

## Start the TotalView Debugger Server

To start a TotalView debugging session over a serial line from the command line, you must first start the TotalView Debugger Server (**tvdsrv**).

Using the console connected to the target machine, start **tvdsrv** and enter the name of the serial port device on the target machine. Here is the syntax of the command you would use:

```
tvdsrv -serial device[:baud=num]
```

where:

<i>device</i>	The name of the serial line device.
<i>num</i>	The serial line's baud rate; if you omit the baud rate, TotalView uses a default value of <b>38400</b>

For example:

```
tvdsrv -serial /dev/com1:baud=38400
```

After it starts, the TotalView Debugger Server will wait for TotalView to establish a connection.

## Starting TotalView on a Serial Line

Start TotalView on the host machine and include the name of the serial line device. The syntax of this command is:

```
totalview -serial device[:baud=num] filename
```

where:

<i>device</i>	The name of the serial line device on the host machine.
<i>num</i>	The serial line's baud rate. If you omit the baud rate, TotalView uses a default value of <b>38400</b> .
<i>filename</i>	The name of the executable file.

For example:

```
totalview -serial /dev/term/a test_pthreads
```

## New Program Window

Here is the procedure for starting a TotalView debugging session over a serial line when you are already in TotalView:

- 1 Start the TotalView Debugger Server. (This is discussed in "Start the TotalView Debugger Server" on page 73).
- 2 Select the **File > New Program** command. TotalView responds by displaying the dialog box shown in Figure 49.  
Type the name of the executable file in the **Executable** field.  
Type the name of the serial line device in the **Serial Line** field.
- 3 Select **OK**.

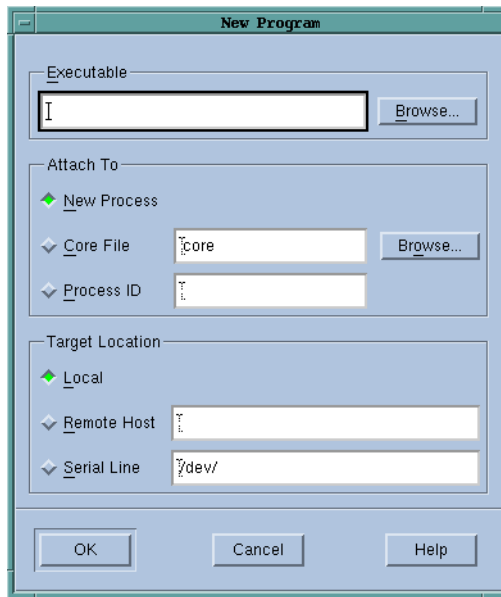


FIGURE 49: File > New Program Dialog Box



## Chapter 5

# Setting Up Parallel Debugging Sessions

This chapter explains how to set up TotalView parallel debugging sessions for applications that use the following parallel execution models. The topics discussed are:

- MPI (and MPICH)
- OpenMP
- ORNL PVM and Compaq DPVM
- SGI “shared memory” (shmem)
- Portland Group HPF

**NOTE** TotalView lets you decide which process you want it to attach. You will find information in “*Attaching to Processes*” on page 117.

## Debugging MPI Applications Overview

You can use TotalView to debug your Message Passing Interface (MPI) programs. With TotalView, you can:

- Automatically acquire processes at startup.
- Attach to a parallel program and automatically acquire the parallel processes.
- Display the message queue state of a process.

Automatic process acquisition at startup is supported for the following MPI implementations:

- MPICH version 1.1.0 or later running on any platform that is supported by both TotalView and MPICH (see “*Debugging MPICH Applications*” on page 78). (You are strongly urged to use a later version of MPICH. Infor-

mation on versions that work with TotalView can be found in the TOTALVIEW PLATFORMS document.)

- Compaq MPI (DMPI) running on Compaq Alpha (see “*Debugging Compaq MPI Applications*” on page 82).
- HP MPI running on HP PA-RISC 1.1 or 2.0 processors (see “*Debugging HP MPI Applications*” on page 83).
- IBM MPI Parallel Environment (PE) running on AIX on RS/6000 and SP (see “*Debugging IBM MPI (PE) Applications*” on page 84).
- SGI MPI running on IRIX on MIPS processors (see “*Debugging SGI MPI Applications*” on page 89).
- QSW RMS running on Compaq AlphaServer SC systems (see “*Debugging QSW RMS Applications*” on page 88).

For more information on message queue display, see “*Displaying the Message Queue Graph*” on page 90.

For tips on debugging parallel applications, see “*Parallel Debugging Tips*” on page 117.

## Debugging MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.1.0 or later on a homogenous collection of machines. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at [www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi). (You are strongly urged to use a later version of MPICH. Information on versions that work with TotalView can be found in the TOTALVIEW PLATFORMS document.)

The MPICH library should use the **ch\_p4**, **ch\_shmem**, **ch\_lfshmem**, or **ch\_mpl** devices. For networks of workstations, **ch\_p4** is the normal default. For shared-memory SMP machines, use **ch\_shmem**. On an IBM SP machine, use the **ch\_mpl** device. The MPICH source distribution includes all of these devices and you can choose one when you configure and build MPICH.

**NOTE** When configuring MPICH, you must ensure that the MPICH library maintains all of the information required by TotalView. Use the `–debug` option with the MPICH `configure` command. In addition, the TotalView Release Notes contains information on patching your MPICH 1.1.0 distribution.

## Starting TotalView on an MPICH Job

Before you can an MPICH job under TotalView’s control, you must have both TotalView and the TotalView server in you path. You can set this up in your login or shell start-up scripts.

To start a job under TotalView’s control, add the `–tv` option to the `mpirun` command:

```
mpirun [ MPICH-arguments ] –tv program [ program-arguments ]
```

For example:

```
mpirun –np 4 –tv sendrecv
```

The MPICH `mpirun` command extracts the value of the `TOTALVIEW` environment variable and then uses its value when starting the first process in the parallel job.

**NOTE** In other contexts, setting this environment variable means that you can use a different TotalView or pass command-line options to TotalView.

For example, here is the C shell command that sets the `TOTALVIEW` environment variable so that `mpirun` will start TotalView using the `–no_stop_all` option:

```
setenv TOTALVIEW "totalview –no_stop_all"
```

TotalView begins by starting the first process of your job, the master process, under its control. You can then set breakpoints, and begin debugging your code.

On the IBM SP machine, the `mpirun` command uses the `poe` command to start an MPI job. While you still must use the MPICH `mpirun` (and its `–tv` option) command to start an MPICH job, the way you start MPICH differs since you are using `poe`. For details of using TotalView with `poe`, see “Starting TotalView on a PE Job” on page 86.

TotalView will automatically acquire the other processes that make up your parallel job. A dialog box will appear that asks if you want to stop the spawned processes. If your answer is **Yes**, you can stop processes as they are initialized. This lets you check their states before they run too far.

TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. Consequently, you do not have to stop them just to set these breakpoints. Next, TotalView updates the Root Window Attached Page to show these newly acquired processes.

## Attaching to an MPICH Job

TotalView allows you to attach to an MPICH application even if it was not started under TotalView's control. Here is the procedure:

- 1 Start TotalView in the normal manner.
- 2 The Root Window Unattached Page displays the processes that are not yet owned, as shown in the following figure.

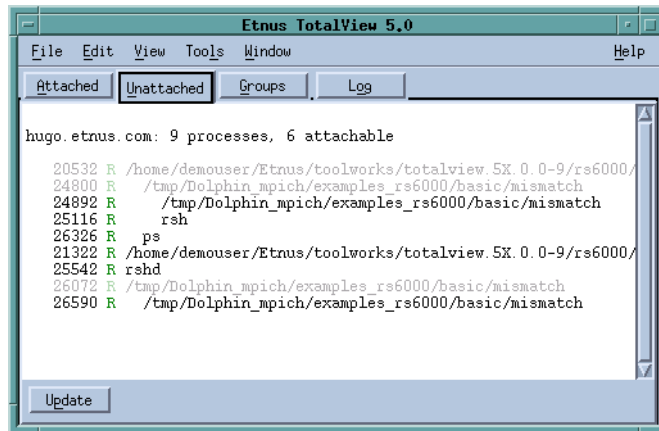


FIGURE 50: Unattached Page

- 3 Attach to the first MPICH process in your workstation cluster by diving into it.

On an IBM SP, attach to the **poe** process that started your job. For details, see “Starting TotalView on a PE Job” on page 86.



Normally, the first MPICH process is the highest process with the correct image name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if you have configured MPICH with **ch\_p4**.
- Additional slave processes if you have configured MPICH with **ch\_shmem** or **ch\_lfshmem**.
- Additional slave processes if you have configured MPICH with **ch\_p4** and have a machine file that places multiple processes on the same machine.

- 4 After you attach to the processes, TotalView asks if you also wish to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, select **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

In some situations, the processes you expect to see may not exist (for example, they may have crashed or exited). TotalView acquires all the processes it can and then warns you if it could not attach to some of them. If you attempt to dive into a process that no longer exists (for example, through the source or target fields of a message state display), TotalView tells you that the process no longer exists.

## MPICH P4 procgroup Files

If you are using MPICH with a P4 **procgroup** file (by using the **-p4pg** option), make sure you use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. If your **procgroup** file contains a different path name that resolves to the same executable, TotalView treats each name as a separate executable, which causes debugging problems.

The following example uses the *same* absolute path name on TotalView's command line and in the **procgroup** file:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun -p4pg p4group -tv /users/smith/mympichexe
```

In this example, TotalView:

- 1 Reads the symbols from **mympichexe** only once.
- 2 Places MPICH processes in the same TotalView share group.
- 3 Names the processes **mympichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If TotalView assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you should check the contents of your **procgrou** file and **mpirun** command line.

## Debugging Compaq MPI Applications

You can debug Compaq MPI applications on the Compaq Alpha platform. To use TotalView with Compaq MPI, you must use Compaq MPI version 1.7 or later.

### Starting TotalView on a Compaq MPI Job

Compaq MPI programs are most often started with the **dmpirun** command. You would use very similar command when starting an MPI program under TotalView's control:

```
totalview dmpirun -a dmpirun-command-line
```

This invokes TotalView and tells it to show you the code for the main program in **dmpirun**. Since you are not usually interested in debugging this code, you can use the **Process > Go** command to let the program run.

The **dmpirun** command runs and starts all of the MPI processes. TotalView acquires them and then asks if you want to stop them all.

**NOTE** Problems can occur while rerunning Compaq MPI programs under TotalView control due to resource allocation issues within Compaq MPI. Consult the Compaq MPI manuals and release notes for information on using **mpiclean** to clean up the MPI system state.

## Attaching to a Compaq MPI Job

To attach to a running Compaq MPI job, attach to the **dmpirun** process that started the job. The procedure for attaching to a **dmpirun** process is the same as the procedure for attaching to other processes. For details, see "Attaching to Processes" on page 39.

After TotalView has attached to the **dmpirun** process, it displays the same dialog as it does with MPICH. (See step 4 on page 81, included in "Attaching to an MPICH Job" on page 80.)

## Debugging HP MPI Applications

You can debug HP MPI applications on a PA-RISC 1.1 or 2.0 processor. To use TotalView with HP MPI, you must use HP MPI version 1.6.

## Starting TotalView on an HP MPI Job

TotalView lets you start an MPI program in three ways:

### **totalview program -a mpi-arguments**

This command tells TotalView to start the MPI process. TotalView will then show you the machine code for the HP MPI **mpirun** executable. Since you are not usually interested in debugging this code, you should let the program run by using the **Process > Go** command.

### **mpirun mpi-arguments -tv -f startup\_file**

This third method tells MPI that it should start TotalView and then start the MPI processes as they are defined within the *startup\_file* script. This script names the processes that will be started. Typically, this file has contents that are similar to:

```
-h localhost -np 1 sendrecv
-h localhost -np 1 sendrecvva
```

In this example, **sendrecv** and **sendrecvva** are two different executable programs. (Your HP MPI documentation describes the contents of this file.)

***mpirun mpi-arguments -tv program***

This command tells MPI that it should start TotalView.

Just before **mpirun** starts the MPI processes, TotalView acquires them and asks if you want to stop the process before it starts executing. If your answer is **yes**, TotalView halts them before they enter the main program. You can then enter breakpoints.

**Attaching to an HP MPI Job**

To attach to a running HP MPI job, attach to the HP MPI **mpirun** process that started the job. The procedure for attaching to an **mpirun** process is the same as the procedure for attaching to any other process. For details, see “*Attaching to Processes*” on page 39.

After TotalView has attached to the HP MPI **mpirun** process, it displays the same dialog as it does with MPICH. (See step 4 on page 81 of “*Attaching to an MPICH Job*” on page 80.)

**Debugging IBM MPI (PE) Applications**

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView’s automatic process acquisition capabilities, you must be running release 2.2 or later of the Parallel Environment for AIX. If you are not running release 2.2, you can run TotalView on release 2.1 if you also load PTF 15.

See “*Displaying the Message Queue Graph*” on page 90 for message queue display information.

**Preparing to Debug a PE Application**

The following sections describe what you must do before TotalView can display a PE application.

## Switch-Based Communication

If you are using switch-based communications (either “IP over the switch” or “user space”) on an SP machine, you must configure your PE debugging session so that TotalView can use “IP over the switch” for communicating with the TotalView Debugger Server (**tvdsrv**), by setting **adaptor\_use** to **shared** and **cpu\_use** to **multiple**, as follows:

- If you are using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
- Always use the following arguments on the **poe** command line:  
`-adaptor_use shared -cpu_use multiple`

If you do not want to set these arguments in the **poe** command line, set the following environment variables before starting **poe**:

```
setenv MP_ADAPTOR_USE shared
setenv MP_CPU_USE multiple
```

When using “IP over the switch,” the default is usually **shared adaptor use** and **multiple cpu use**; to be safe, set it explicitly by using one of these techniques.

When you are using switch-based communications, you must run TotalView on one of the SP or SP2 nodes. Since TotalView uses “IP over the switch” in this case, you cannot run TotalView on an RS/6000 workstation.

## Remote Login

You must be able to perform a remote login using the **rsh** command. You will also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Debugger Server by using the **rsh** command with the switch host name of the node.

## Timeout

TotalView automatically sets the **timeout** value to 600 seconds. If you receive communications timeouts, you can set the value higher. For example:

```
setenv MP_TIMEOUT 1200
```

**NOTE** The timeout value cannot be set using the **poe** command line.

## Starting TotalView on a PE Job

Parallel Environment (PE) programs are run from the command line by using the following syntax:

```
program [ arguments ] [ PE_arguments ]
```

They can also be run by using the **poe** command:

```
poe program [ arguments ] [ PE_arguments ]
```

However, if you start TotalView on a PE application, you must start the **poe** executable as TotalView's target. The syntax for this is:

```
totalview poe -a program [ arguments ] [ PE_arguments ]
```

For example:

```
totalview poe -a sendrecv 500 -rmpool 1
```

## Setting Breakpoints

After TotalView is running, you can start the **poe** process; this process will then start the program's parallel processes. Next, you will need to use the **Process > Go** command. TotalView responds by displaying a dialog box that asks if you want to stop the parallel tasks.

If you want to set breakpoints in your code at this point, answer **Yes** to stop the processes. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. After a Process Window for the first parallel task appears, you can set breakpoints and control the parallel tasks in the same way as any process controlled by TotalView.

If you have already set and saved breakpoints in a file and want to reload the file, answer **No**. After TotalView loads these breakpoints, the parallel tasks continue to run.

## Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks with the Process Window's **Group > Go** command.

**NOTE** No parallel tasks will reach the first line of code in your **main** routine until all parallel tasks start.

You should be very cautious in placing breakpoints at or before a line that calls **MPI\_Init()** or **MPL\_Init()** because timeouts can occur during initialization. Once you allow the parallel processes to proceed into the **MPI\_Init()** or **MPL\_Init()** call, you should allow all of the parallel processes to proceed through it within a short time. For more information on this, see “*Avoid unwanted timeouts*” on page 122.

## Attaching to a PE Job

To take full advantage of TotalView’s **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on its various nodes. This set of acquired processes will include the processes you want to debug.

### Attaching from a Node Running poe

Here’s the procedure for attaching TotalView to **poe** from the node running **poe**.

- 1** Start TotalView in the directory of the debug target.  
If you cannot start TotalView in the debug target directory, you can start TotalView by editing the TotalView Debugger Server (**tvdsrv**) command line before attaching to **poe**. See “*Single Process Server Launch Command*” on page 66.
- 2** Within the Root Window’s Unattached Page, find the **poe** process list, and attach to it by diving into it. When necessary, TotalView launches TotalView Debugger Servers. TotalView will also update the Root Window’s Attached Page and open a Process Window for the **poe** process.
- 3** Locate the process you want to debug and dive on it. TotalView responds by opening a Process Window for it.

If source code files are available but are not displayed in the Source Pane, you may have not told TotalView where these files reside. You can fix this by invoking the **File > Search Path** command to add directories to your search path.

### Attaching from a Node Not Running poe

To attach TotalView to **poe** from a node not running **poe**, follow the same procedures as in attaching from a node running **poe**, except, since you did not run TotalView from the node running **poe** (the start-up node), you will not be able to see **poe** on the process list in your Root Window's Attached Page and you will not be able to start it by diving into it.

The procedure for placing **poe** within this list is:

- 1 Connect TotalView to the start-up node. For details, see "Starting the TotalView Debugger Server" on page 61 and "Attaching to Processes" on page 39.
- 2 Select the Root Window's Unattached Page, and then invoke the **Window > Update** command.
- 3 Look for the process named **poe** and continue as if attaching from a node running **poe**.

## Debugging QSW RMS Applications

TotalView supports automatic process acquisition on AlphaServer SC systems that use Quadrics' RMS resource management system with the QSW switch technology.

**NOTE** Message queue display is only supported if you are running version 1, patch 2 or later of AlphaServer SC.

### Starting TotalView on an RMS Job

To start a parallel job under TotalView's control, use TotalView as though you were debugging the **prun** command:

```
totalview prun -a prun-command-line
```

TotalView starts up and shows you the machine code for RMS **prun**. Since you are not usually interested in debugging this code, use the **Process > Go** command to let the program run.

The RMS **prun** command executes and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you



do stop them, TotalView halts them before they enter the main program. You can then enter breakpoints.

## Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** processes the same way you attach to other processes. For details on attaching to processes, see “*Attaching to Processes*” on page 39.

Once you have attached to the RMS **prun** process, TotalView displays the dialog as it does with MPICH. (See step 4. on page 81 of “*Attaching to an MPICH Job*” on page 80.)

## Debugging SGI MPI Applications

TotalView can acquire processes started by SGI MPI version 3.1, which is part of the Message Passing Toolkit (MPT) 1.2 package.

Message queue display is supported by release 1.3 of the Message Passing Toolkit. See “*Displaying the Message Queue Graph*” on page 90 for message queue display.

## Starting TotalView on a SGI MPI Job

SGI MPI programs are normally started with the **mpirun** command. You would use a similar command to start an MPI program under TotalView’s control:

```
totalview mpirun -a mpirun-command-line
```

This invokes TotalView and tells it to show you the machine code for SGI MPI **mpirun**. Since you are not usually interested in debugging this code, use the **Process > Go** command to let the program run.

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you

answer *yes*, TotalView halts them before they enter the main program. You can then enter breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message showing the name of the array and the value of the array services handle (**ash**) to which it is attaching.

## Attaching to an SGI MPI Job

To attach to a running SGI MPI job, attach to the SGI MPI **mpirun** process that started the job. The procedure for attaching to an **mpirun** process is the same as the procedure for attaching to any other process. For details, see “*Attaching to Processes*” on page 39.

Once you have attached to the SGI MPI **mpirun** process, TotalView displays the same dialog as it does with MPICH. (See step 4 on page 81 of “*Attaching to an MPICH Job*” on page 80.)

## Displaying the Message Queue Graph

TotalView can graphically display your MPI program’s message queue state. If you select the Process Window’s **Tools > Message Queue Graph** command, TotalView displays a window with a large empty area. After you select the ranks to be monitored, the kind of messages, and message states, TotalView updates this window to show the current queue state. See Figure 51 on page 91 for a sample window.

The numbers within the boxes indicate a process’s rank. Diving on a box tells TotalView that it should open a Process Window for that process.

The numbers next to the arrows indicate the number of messages when TotalView created the graph. Diving on the arrow tells TotalView that it should display its **Tools > Message Queue** Window, which will have detailed information about the messages.

This graph shows you the state of your program at a particular instant. Selecting the **Display** button tells TotalView that it should update the display.

While you can use this window in many ways, here are some to consider:

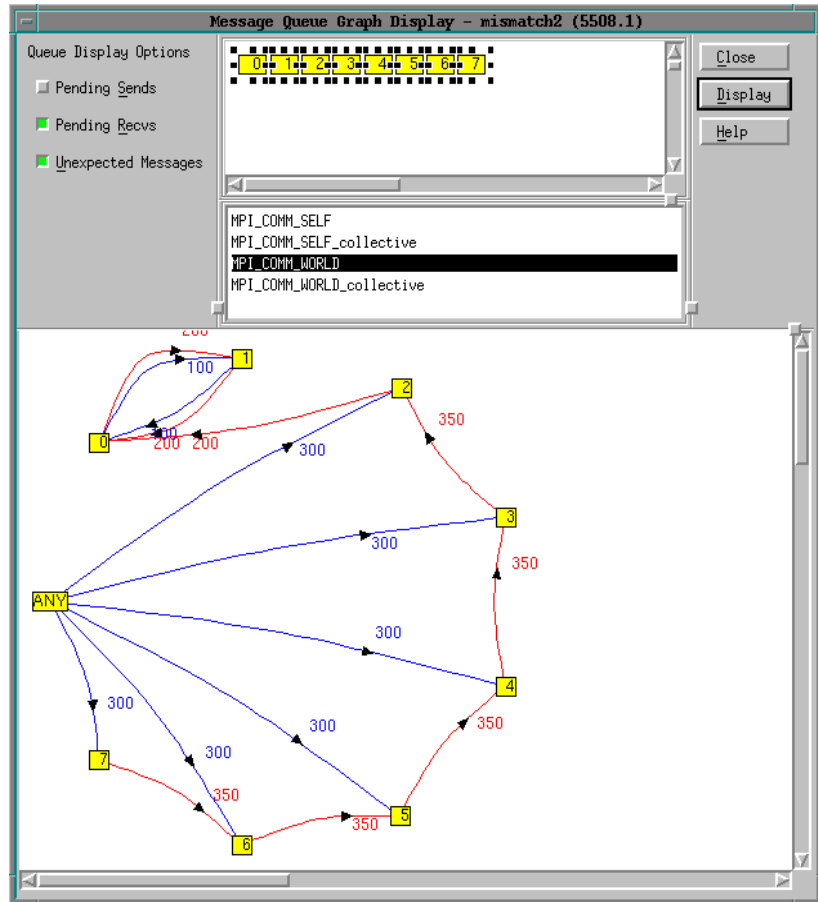


FIGURE 51: Tools &gt; Message Queue Graph Window

- Pending messages often indicate that a process cannot keep up with the amount of work it is expected to perform. These messages indicate places where you may be able to improve your program's efficiency.
- Unexpected messages can indicate that something is wrong with your program because the receiving process does not know how to process the message.
- After a while, the shape of the graph tends to tell you something about how your program is executing. If something does not look right, you might want to determine why it looks different.

- You can change the shape of the graph by dragging either nodes or the arrows. This is often useful when you are comparing sets of nodes and their messages with one another. TotalView does not remember the places to which you have dragged the nodes and arrows. This means that if you select the **Display** button after you arrange the graph, your changes are lost.

## Displaying the Message Queue

The **Tools > Message Queue** dialog box displays the your MPI program's message queue state. This can be useful when you need to find the cause of a deadlock.

To use the message queue display feature, you must be using one of the following versions of MPI:

- MPICH version 1.1.0 or later.
- Compaq Alpha MPI (DMPI) version 1.7.
- HP HP-UX version 1.6.
- IBM MPI Parallel Environment (PE) version 2.3 or 2.4; but only for programs using the threaded IBM MPI libraries. MQD is not available with earlier releases, or with the non-thread-safe version of the IBM MPI library. Therefore, to use TotalView MQD with IBM MPI applications, you must compile and link your code using the **mpcc\_r**, **mpxlf\_r**, or **mpxlf90\_r** compilers.
- For the SGI MPI TotalView message queue display, you must obtain the Message Passing Toolkit (MPT) release 1.3 or later. Check with SGI for availability.

## Message Queue Display Overview

After an MPI process returns from the call to **MPI\_Init()**, you can display the internal state of the MPI library by selecting the **Tools > Message Queue** command. The information is shown in Figure 52.

This page displays the state of the process's MPI communicators. In some MPI implementations such as MPICH, user-visible communicators are implemented as two internal communicator structures, one for point-to-point and the other for collective operations. TotalView displays both.

**NOTE** You cannot edit any of the fields in the Message Queue Window.

The contents of the Message Queue dialog box are only valid when a process is stopped. (See Figure 52.)

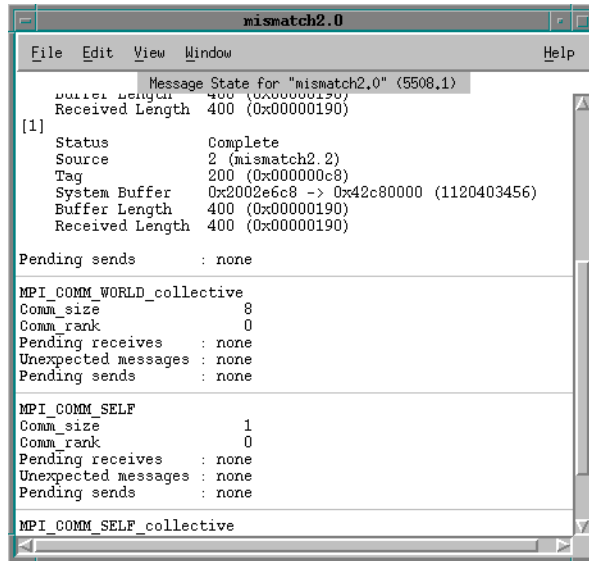


FIGURE 52: Message Queue Window

## Message Operations

For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has an index value displayed in brackets ([*n*]). The online help for this window contains a description of the fields that can be displayed.

## MPI Process Diving

To display more detail, you can dive into certain fields in the Message Queue dialog box. When you dive into a process field, TotalView does one of the following:

- Raises its Process Window if it exists.
- Sets the focus to an existing Process Window on the requested process.
- If a Process Window does not exist, creates a new one for the process.

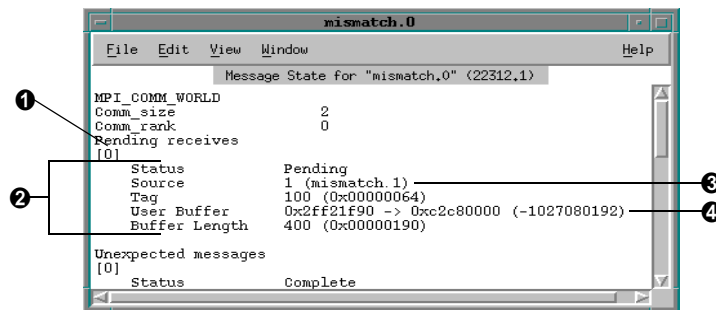
## MPI Buffer Diving

When you dive into the buffer fields, TotalView opens a Variable Window. It also guesses what the correct format for the data should be based on the buffer's length and the data's alignment. If TotalView guesses incorrectly, you can edit the type field in the Variable Window.

**NOTE** TotalView does not set the buffer type using the MPI data type.

## Pending Receive Operations

TotalView displays each pending receive operation in the **Pending receives** list. The following figure shows an example of an MPICH pending receive operation.



- ❶ Operation index
- ❷ One receive operation
- ❸ Diving here displays a Process Window
- ❹ Diving here displays a Variable Window

FIGURE 53: Message Queue Window Showing Pending Receive Operation

**NOTE** TotalView displays all of the receive operations that are maintained by the IBM MPI library. You should set the environment variable `MP_EUIDEVELOP` to the value `DEBUG` if you want blocking operations to be visible; otherwise, only non-blocking operations are maintained. For more details on the `MP_EUIDEVELOP` environment variable, consult the *IBM Parallel Environment Operations and Use* manual.

## Unexpected Messages

The **Unexpected messages** portion of the **Message Queue** Window shows information for retrieved and enqueued messages which are not yet matched with a receive operation.

Some MPI libraries such as MPICH 1.1.1 only retrieve already received messages as a side effect of calls to functions such as **MPI\_Recv()** or **MPI\_Iprobe()**. (In other words, while some versions of MPI may know about the message, the message may not yet be in a queue.) This means that TotalView cannot list a message until after the destination process makes a call that retrieves it.

## Pending Send Operations

TotalView displays each pending send operations in the **Pending sends** list.

MPICH does not normally keep information about pending send operations. However, when you configure MPICH, you can tell it to maintain a list of them. Start your program under TotalView's control and use **mpirun's -ksq**, or **-KeepSendQueue** if MPICH to see this kept.

Depending on the device for which MPICH was configured, blocking send operations may or may not be visible. However, if they are not displayed, you can see that these operations occurred because the call is in the stack backtrace.

If you attach to an MPI program that is not maintaining send queue information, TotalView displays the following message:

Pending sends : no information available

## MPI Debugging Troubleshooting

If you cannot successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView? The MPICH code contains some useful scripts that let you verify that you can start remote processes on all of the machines in your machines file. (See **tstmachines** in **mpich/util**.)
- You will not get a message queue display if you get the following warning:  
The symbols and types in the MPICH library used by TotalView to extract the message queues are not as expected in the image **<<your image name>>**. This is probably an MPICH version or configuration problem.

You need to check that you are using MPICH 1.1.0 or later and that you have configured it with the `-debug` option. (You can check this by looking in the `config.status` file at the root of the MPICH directory tree).

- Does the TotalView Debugger Server (`tvdsrv`) fail to start? `tvdsrv` must be on your `PATH` when you log in. Remember that `rsh` is being used to start the server, and it does not pass your current environment to the process you started remotely.
- Make sure you have the correct MPI version and have applied any required patches. See the TOTALVIEW RELEASE NOTES for up-to-date information.
- Under some circumstances, MPICH kills TotalView with the `SIGINT` signal. You can see this behavior when restarting an MPICH job using the `Group > Delete` command. If TotalView exits and is terminated abnormally with a `Killed` message, try setting the TotalView `-ignore_control_c` command-line option. For example:

```
setenv TOTALVIEW "totalview -ignore_control_c"
mpirun -tv /users/smith/mympichexe
```

## Debugging OpenMP Applications

TotalView provides explicit support for many OpenMP C and Fortran compilers. The compilers and architectures that we support are listed in the TOTALVIEW PLATFORMS document and our Web site.

Here are some of the features that TotalView supports:

- Source-level debugging of the original OpenMP code.
- The ability to plant breakpoints throughout the OpenMP code, including lines that are executed in parallel.
- Visibility of OpenMP worker threads.
- Access to **SHARED** and **PRIVATE** variables in OpenMP **PARALLEL** code.
- A stack-back link token in worker threads' stacks so that you can find their master stack.
- Access to OMP **THREADPRIVATE** data in code compiled by the IBM and Compaq compilers.

The code examples used in this section are included in the TotalView distribution in the `examples/omp_simple_f` file.



**NOTE** On the SGI IRIX platform, you must use the MIPSpro 7.3 compiler or later to debug OpenMP.

## Debugging an OpenMP Program

Debugging OpenMP code is very similar to debugging multithreaded code, differing only in that the OpenMP compiler makes the following special code transformations:

- The most visible transformation is *outlining*. The compiler pulls the body of a parallel region out of the original routine and places it into an *outlined routine*. In some cases, the compiler will generate multiple outlined routines from a single parallel region. This allows multiple threads to execute the parallel region.  
The outlined routine's name is based on the original routine's name.
- The compiler inserts calls to the OpenMP runtime library.
- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables are maintained in the master thread's original routine, and private variables are maintained in the outlined routine.
- The master thread creates threads to share the workload. As the master thread begins to execute a parallel region in the OpenMP code, it creates the worker threads, dispatches them to the outlined routine, and then calls the outlined routine itself.

TotalView makes these transformations visible in the debugging session. Here are some things you should know:

- The compiler may generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions.  
If you set a breakpoint on a source line that results in multiple outlined routines, TotalView displays its **Ambiguous Line** dialog box that lets you to select the function name. In most cases, you will select the **All** button to operate on all instances of the outlined functions.
- You cannot single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and allow the process to run to it. Once inside a parallel region, you can single step within it.
- OpenMP programs are multithreaded programs, so the rules for debugging multithreaded programs apply.

Figure 54 shows a sample OpenMP debugging session.

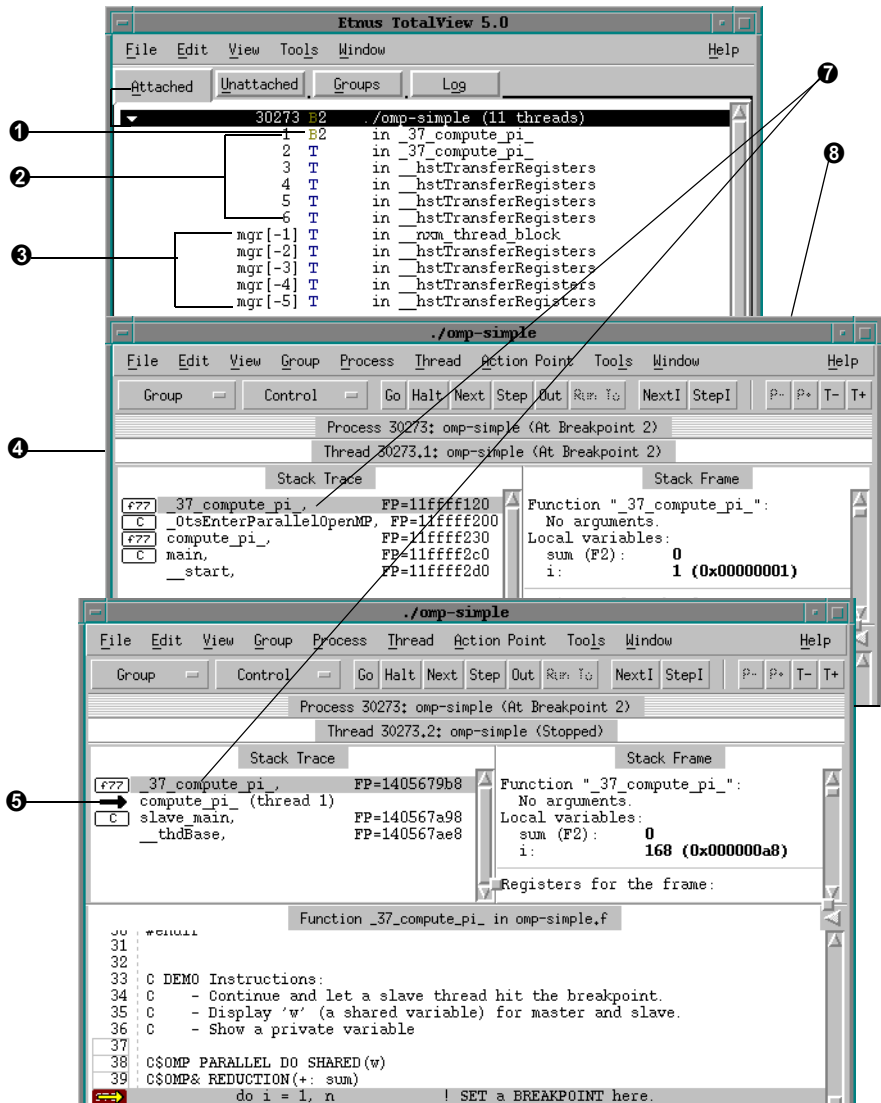
The following list contains information on platform differences:

- On Compaq Tru64 UNIX, the OpenMP threads are implemented by the compiler as **pthread**s, and on SGI IRIX as **sprocs**. TotalView shows the threads' logical and/or system thread ID, not the OpenMP thread number.
- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.
- In Compaq Tru64 UNIX, the system manager threads have a negative thread ID; as they do not take part in your OpenMP program, you should never touch them.
- SGI OpenMP uses the **SIGTERM** signal to terminate threads. Because TotalView stops a process when the process receives a **SIGTERM**, the OpenMP process does not terminate. If you want the OpenMP process to terminate instead of stop, set the default action for the **SIGTERM** signal to *Resend*.
- When the OpenMP master thread is stopped in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
  - The outlined routine called from.
  - The OpenMP runtime library called from.
  - The original routine (containing the parallel region).
- When the OpenMP worker threads are stopped in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
  - Outlined routine called from the special stack parent token line.
  - The OpenMP runtime library called from.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.

## OpenMP Private and Shared Variables

TotalView allows you to view both OpenMP private and shared variables.

OpenMP private variables are maintained in the outlined routine, and are stored by the compiler like local variables. See “*Displaying Local Variables and Registers*” on page 153. However, OpenMP shared variables are maintained in the master thread's original routine stack frame.



- |                                  |   |
|----------------------------------|---|
| ① OpenMP master thread           | ⑤ “Original” routine name                           |
| ② OpenMP worker threads          | ⑥ Stack parent token. Select or dive to view master |
| ③ Manager threads (do not touch) | ⑦ “Outlined” routine name                           |
| ④ Master thread window           | ⑧ Worker thread process                             |

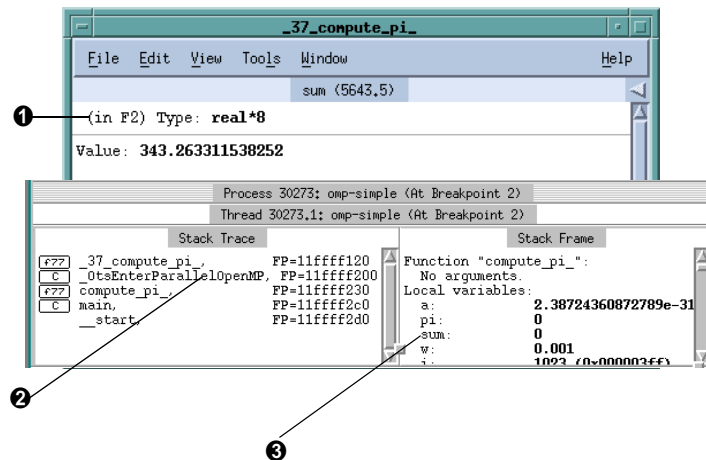
FIGURE 54: Sample OpenMP Debugging Session

TotalView allows you to display shared variables through a Process Window focused on the OpenMP master thread or through one of the OpenMP worker threads.

Here is how you display an OpenMP shared variable:

- 1 Select the outlined routine in the Stack Trace Pane; or select the original routine stack frame in the OpenMP master thread.
- 2 Dive on the variable name, or select the **View > Lookup Variable** command. When prompted, enter the variable name.

TotalView will open a Variable Window displaying the value of the OpenMP shared variable, as shown in Figure 55.



- 1 OpenMP shared variables have master thread's context
- 2 Original routine's stack frame selected
- 3 Stack Frame Pane includes shared variables

FIGURE 55: OpenMP Shared Variable

Shared variables are stored on the OpenMP master thread's stack. When displaying shared variables in OpenMP worker threads, TotalView uses the stack context of the OpenMP master thread to find the shared variable. TotalView uses the OpenMP master thread's context in the resulting Variable Window to display the shared variable.

You can also view OpenMP shared variables in the Stack Frame Pane by selecting the original routine stack frame in the OpenMP master thread, or by selecting the stack parent token line in the Stack Trace Pane of OpenMP worker threads, as shown in Figure 55.

## OpenMP THREADPRIVATE Common Blocks

The Compaq Tru64 UNIX OpenMP and SGI IRIX compilers implement OpenMP THREADPRIVATE common blocks by using the thread local storage system facility. This facility stores a variable declared in OpenMP THREADPRIVATE common blocks at different memory locations for each thread in an OpenMP process. This allows the variable to have different values in each thread.

Here's how you can view a variable in an OpenMP THREADPRIVATE common block, or the OpenMP THREADPRIVATE common block itself:

- 1 In the Threads Pane of the Process Window, select the thread containing the private copy of the variable or common block you would like to view.
- 2 In the Stack Trace Pane of the Process Window, select the stack frame that will allow you to access OpenMP THREADPRIVATE common block variable. You can select either the outlined routine or the original routine for an OpenMP master thread. You must, however, select the outlined routine for an OpenMP worker thread.
- 3 From the Process Window, dive on the variable name or common block name. Or, select the **View > Lookup Variable** command. When prompted, enter the name of the variable or common block. You may need to append an underscore ( `_` ) after the common block name. TotalView opens a Variable Window displaying the value of the variable or common block for the selected thread.  
See "Displaying Variable Windows" on page 153 for more information on displaying variables.
- 4 To view OpenMP THREADPRIVATE common blocks or variables across all threads, you can use the Variable Window's **View > Laminate Threads** command. See "Displaying a Variable in All Processes or Threads" on page 196.

Figure 56 shows Variable Windows displaying an OpenMP THREADPRIVATE common blocks. Because the Variable Window has the same thread context as the Process Window from which it was created, the title bar patterns for the same thread match. In the laminated views, the values of the common block across all threads are displayed.

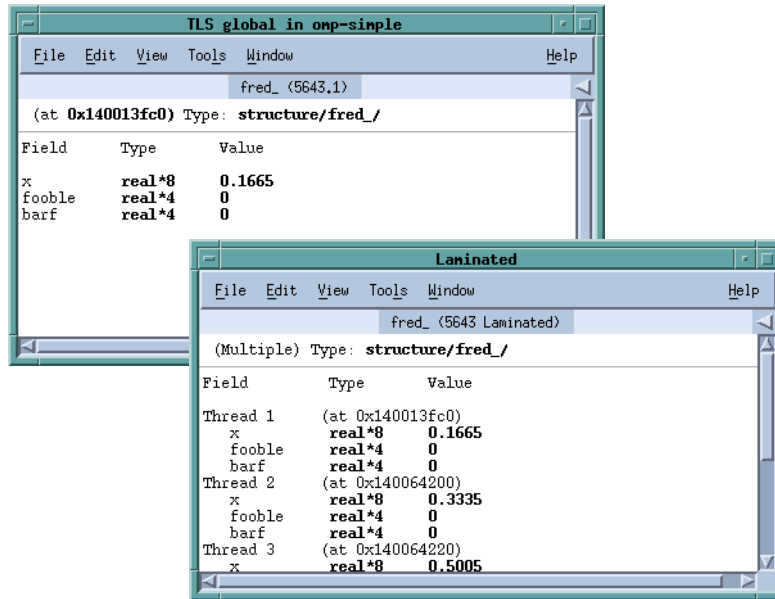


FIGURE 56: OpenMP THREADPRIVATE Common Block Variables

## OpenMP Stack Parent Token Line

TotalView inserts a special stack parent token line in the Stack Trace Pane of OpenMP worker threads when they are stopped in an outlined routine.

When you select or dive on the stack parent token line, the Process Window switches to the OpenMP master thread, allowing you to see the stack context of the OpenMP worker thread's routine. This context includes the OpenMP shared variables. (See Figure 57.)

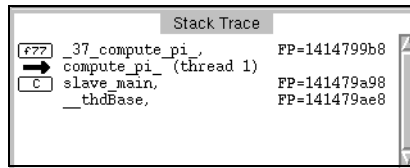


FIGURE 57: OpenMP Stack Parent Token Line

## Debugging PVM and DPVM Applications

You can debug applications that use the Parallel Virtual Machine (PVM) library or the Compaq Tru64 UNIX Parallel Virtual Machine (DPVM) library with TotalView on some platforms. TotalView supports ORNL PVM 3.4.1 on the Compaq Alpha, Hewlett-Packard, Sun 5, RS/6000, and SGI IRIX platforms and DPVM 1.4 or later on the Compaq Alpha platform.

**NOTE** See the TotalView Platforms document for the most up-to-date information regarding your PVM or DPVM software.

For tips on debugging parallel applications, see “Parallel Debugging Tips” on page 117.

## Supporting Multiple Sessions

When you debug a PVM or DPVM application, TotalView becomes a PVM tasker. This lets it establish a debugging context for your session. You can run:

- One TotalView PVM or DPVM debugging session for a user and for an architecture; that is, different users cannot interfere with each other on the same machine or same machine architecture.

One user can start TotalView to debug the same PVM or DPVM application on different machine architectures. However, a single user cannot have multiple instances of TotalView debugging the same PVM or DPVM session on a single machine architecture.

For example, suppose you start a PVM session on Sun 5 and Compaq Alpha machines. You must start two TotalView sessions: one on the Sun 5 machine to debug the Sun 5 portion of the PVM session, and one on the Compaq Alpha machine to debug the Compaq Alpha portion of the PVM

session. These two TotalView sessions are separate and do not interfere with one another.

- Similarly, in one TotalView session, you can run either a PVM application or a DPVM application, but not both. However, if you run TotalView on a Compaq Alpha, you can have two TotalView sessions: one debugging PVM and one debugging DPVM.

## Setting Up ORNL PVM Debugging

To enable PVM, create a symbolic link from the PVM **bin** directory (which is `$HOME/pvm3/bin/$PVM_ARCH/tvdsvr`) to the TotalView Debugger Server (**tvdsvr**). With this link in place, TotalView can use the `pvm_spawn()` call to spawn the **tvdsvr** tasks.

For example, if **tvdsvr** is installed in the `/opt/totalview/bin` directory, enter the following command:

```
In -s /opt/totalview/bin/tvdsvr \
    $HOME/pvm3/bin/$PVM_ARCH/tvdsvr
```

If the symbolic link does not exist, TotalView cannot spawn **tvdsvr**. If this happens, TotalView displays the following error:

Error spawning TotalView Debugger Server: No such file

## Starting an ORNL PVM Session

Start the ORNL PVM daemon process before you start TotalView. See the ORNL PVM documentation for information about the PVM daemon process and console program.

- 1 Use the **pvm** command to start a PVM console session—this command starts the PVM daemon. If PVM is not running when you start TotalView (with PVM support enabled), TotalView exits with the following message:  
 Fatal error: Error enrolling as PVM task: pvm error
- 2 If your application uses groups, start the **pvmgs** process before starting TotalView. PVM groups are unrelated to TotalView process groups. For information about TotalView process groups, refer to “*Examining Groups*” on page 135.



- 3 Enable PVM support in TotalView using an X resource; see **totalview\*pvmDebugging** on page 281. You need to restart TotalView after setting this new resource. For more information, refer to “X Resources” on page 275.

As an alternative, you can use command-line options to the **totalview** command. For example:

```

-pvm           which enables PVM support
-no_pvm       which disables PVM support

```

The command-line options override the X resource. For more information on, refer to “TotalView Command Syntax” on page 289.

- 4 Set the TotalView directory search path to include the PVM directories. This directory list must include those needed to find both executable and source files. The directories you use will vary, but should always contain the current directory and your home directory.

You can set the directory search path by using an X resource or the **File > Search Directory** command. Refer to **totalview\*searchPath** on page 281 and “Setting Search Paths” on page 48 for more information.

For example, to debug the PVM examples, you can place the following directories in your search path:

```

$HOME
$PVM_ROOT/xep
$PVM_ROOT/xep/$PVM_ARCH
$PVM_ROOT/src
$PVM_ROOT/src/$PVM_ARCH
$PVM_ROOT/bin/$PVM_ARCH
$PVM_ROOT/examples
$PVM_ROOT/examples/$PVM_ARCH
$PVM_ROOT/gexamples
$PVM_ROOT/gexamples/$PVM_ARCH

```

- 5 Verify that the action taken by TotalView for the **SIGTERM** signal is appropriate. (You can examine the current action by using the Process Window’s **File > Signals** command. Refer to “Handling Signals” on page 45 for more information.)

PVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the OpenMP process is not terminated. If you want the PVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

Continue with “Automatically Acquiring PVM/DPVM Processes” on page 107.

## Starting a DPVM Session

DPVM requires no additional user configuration. However, you must start the DPVM daemon before you start TotalView. See the DPVM documentation for information about the DPVM daemon and console program.

- 1 Use the **dpvm** command to start a DPVM console session; starting the session also starts the DPVM daemon. If DPVM is not running when you start TotalView (with DPVM support enabled), TotalView exits with the following message:

Fatal error: Error enrolling as DPVM task: dpvm error

- 2 You can enable DPVM support in two ways. The first uses an X resource; see **totalview\*DPVMDebugging** on page 277. You will need to restart TotalView after setting (or resetting) an X resource.

As an alternative, you can use command-line options to the **totalview** command. For example:

**-dpvm**            *which enables DPVM support.*  
**-no\_dpvm**        *which disables DPVM support*

The command-line options override the X resource. For more information on the **totalview** command, refer to "TotalView Command Syntax" on page 289.

- 3 Verify that the default action taken by TotalView for the **SIGTERM** signal is appropriate. You can examine the default actions with the Process Window's **File > Signals** command in TotalView. Refer to "Handling Signals" on page 45 for more information.

DPVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the OpenMP process is not terminated. If you want the DPVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

If you enable PVM support using X resources, and you wish to use DPVM, you must use both **-no\_pvm** and **-dpvm** command-line options when you start TotalView. Similarly, when enabling DPVM support using an X resource, you can use the **-no\_dpvm** and **-pvm** command-line options to debug PVM.

**NOTE** Do not use X resources to start both PVM and DPVM.

## Automatically Acquiring PVM/DPVM Processes

This section describes how TotalView automatically acquires PVM and DPVM processes in a PVM or DPVM debugging session. Specifically, TotalView uses the PVM tasker to intercept `pvm_spawn()` calls.

When you start TotalView as part of a PVM or DPVM debugging session, it takes the following actions:

- TotalView checks to make sure there are no other PVM or DPVM taskers running. If TotalView finds a tasker on any host that it is debugging, it displays the following message and then exits:
 

**Fatal error: A PVM tasker is already running on host '*host*'**
- TotalView finds all the hosts in the PVM or DPVM configuration. Using the `pvm_spawn()` call, TotalView starts a TotalView Debugger Server (`tvdsrv`) on each remote host that has the same architecture type as the host on which TotalView is running. It tells you it has started a debugger server by printing:

**Spawning TotalView Debugger Server onto PVM host '*host*'**

If you add a host with a compatible machine architecture to your PVM or DPVM debugging session after you start TotalView, TotalView automatically starts a debugger server on that host.

After all debugger servers are running, TotalView will intercept every PVM or DPVM task created with the `pvm_spawn()` call on hosts that are part of the debugging session. If a PVM or DPVM task is created on a host with a different machine architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, it takes the following actions:

- 1 TotalView reads the symbol table of the spawned executable.
- 2 If a saved breakpoints file for the executable exists and you have enabled automatic loading of breakpoints, TotalView loads breakpoints for the process.
- 3 TotalView asks if you want to stop the process before it enters the `main()` routine.

If you answer **Yes**, TotalView stops the process before it enters `main()` (that is, before it executes any user code). This allows you to set break-

points in the spawned process before any user code executes. On most machines, TotalView stops a process in the **start()** routine of the **crt0.o** module if it is statically linked. If the process is dynamically linked, the debugger stops it just after it finishes running the dynamic linker. Because the Process Window displays assembler instructions, you will need to use the **View > Lookup Function** command to display the source code for the **main()** routine. For more information on this command, refer to “*Finding the Source Code for Functions*” on page 127.

## Attaching to PVM/DPVM Tasks

You can attach to a PVM or DPVM task if the task meets the following criteria:

- The machine architecture on which the task is running is the same as the machine architecture on which TotalView is running.
- The task must be created. (This is indicated when flag 4 is set in the PVM Tasks and Configuration Window.)
- The task must not be a PVM tasker. If flag 400 is clear in the PVM Tasks and Configuration Window, the process is a tasker.
- The executable name must be known. If the executable name is listed as a dash (–), TotalView cannot determine the name of the executable. (This can occur if a task was not created with the **pvm\_spawn()** call.)

To attach to a PVM or DPVM task, complete the following steps:

- 1 Select **Tools > PVM Tasks** command from the TotalView Root Window.

The PVM Tasks and Configuration Window is displayed, as shown in Figure 58. This window displays current information about PVM tasks and hosts—TotalView automatically updates this information as it receives events from PVM.

Since PVM does not always generate an event that allows TotalView to update this window, you should use the **Windows > Update** command to ensure that you are seeing the most current information.

For example, you can attach to the tasks named **xep** and **mtile** in the following figure because flag 4 is set. In contrast, you cannot attach to the **tvdsrv** and – (dash) executables because flag 400 is set.

- 2 Dive on a task entry that meets the criteria for attaching to tasks. TotalView attaches to the task.

- 3** If the task to which you attached has related tasks that can be debugged, TotalView asks if you want to attach to these related tasks. If you answer **Yes**, TotalView attaches to them. If you answer **No**, it only attaches to the task you dove on.

After attaching to a task, TotalView looks for attached tasks that are related to the this task; if there are related tasks, TotalView places them in the same control group. If TotalView is already attached to a task you dove on, it simply opens and raises the Process Window for the task. (See Figure 58.)

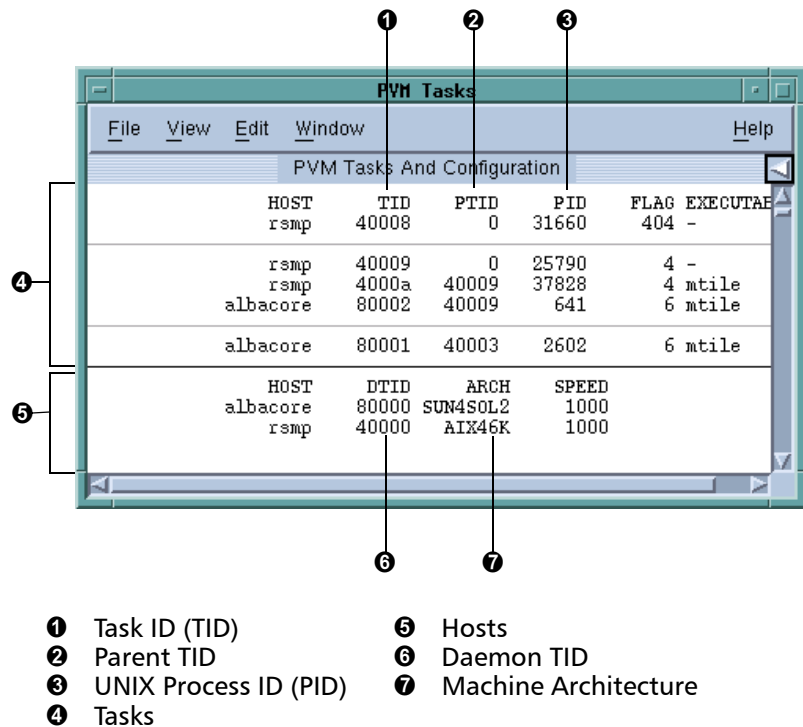


FIGURE 58: PVM Tasks and Configuration Window

## Reserved Message Tags

TotalView uses PVM message tags in the range 0xDEB0 through 0xDEBF to communicate with PVM daemons and the TotalView Debugger Server. Avoid sending messages that use these reserved tags.

## Debugging Dynamic Libraries

If the machines in your PVM debugging session are running different versions of the same operating system, the dynamic libraries can vary from machine to machine. If this is the case, you may see strange stack back-trace results when your program is executing inside a dynamic library. To eliminate this problem, make sure all of the hosts in your PVM configuration are running the same version of the operating system and have the same dynamic libraries installed. As an alternative, you can statically link your programs.

## Cleanup of Processes

The **pvmgs** process registers its task ID in the PVM database. If the **pvmgs** process is terminated, the **pvm\_joingroup()** routine hangs because PVM does not clean up the database. If this happens, you must terminate and then restart the PVM daemon.

TotalView attempts to clean up the TotalView Debugger Server daemons (**tvdsvr**), which also act as taskers. If some of these processes do not terminate, you must manually terminate them.

## Shared Memory Code

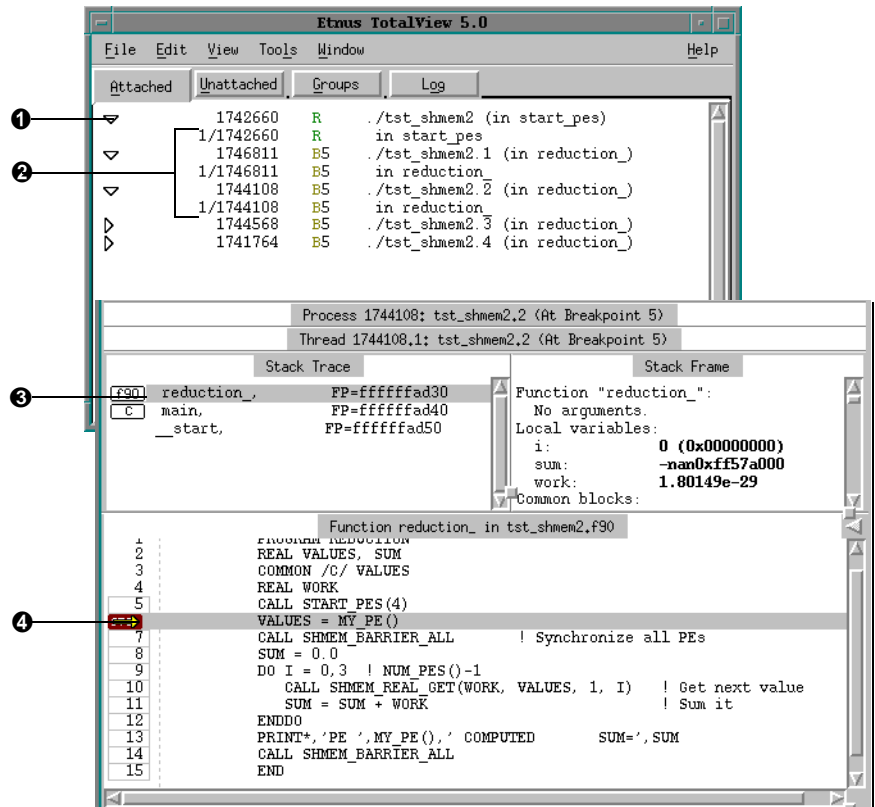
TotalView supports the SGI IRIX logically shared, distributed memory access (SHMEM) library.

To debug a SHMEM program, follow these steps:

- 1 Link it with the **dbfork** library. See “*Linking with the dbfork Library*” on page 317.
- 2 Start TotalView on your program. See Chapter 3, “*Setting Up a Debugging Session*” on page 33.
- 3 Set at least one breakpoint after the call to the **start\_pes()** SHMEM routine. (This is illustrated in Figure 59.)

**NOTE** You cannot single-step over the call to **start\_pes()**.

The call to **start\_pes()** creates new worker processes that return from the **start\_pes()** call and execute the remainder of your program. The original



- ① SHMEM starter process
- ② SHMEM worker processes
- ③ Select a worker process in the Root Window
- ④ Set breakpoint *after* the call to start\_pes()

FIGURE 59: SHMEM Sample Session

process never returns from `start_pes()`, but instead stays in that routine, waiting for the worker processes it created to terminate

## Debugging Portland Group, Inc., HPF Applications

TotalView lets you debug High Performance Fortran (HPF) code compiled with the Portland Group, Inc., HPF (PGI HPF) compiler at the source level.

**NOTE** Debugging PGI HPF programs requires a separate TotalView license.

TotalView supports the following platforms:

- IBM RS/6000 and SP AIX 4.x
- SGI MIPS IRIX 6.x, for programs compiled with -64 only
- Sun Sparc SunOS 5 (Solaris 2.x)

See the TOTALVIEW PLATFORMS document for supported PGI HPF runtime configurations.

In addition to normal TotalView features, the TotalView PGI HPF support allows the following:

- Source-level display of HPF code.
- Source-level breakpoints in HPF code.
- You can update replicated scalar variables in all processes by updating the value in any process.
- Display of distributed arrays, with optional display of the owning processor.
- Display of the distribution of distributed arrays, for instance, onto which node a particular element of a distributed array is mapped.
- Visualization of distributed arrays.
- Automatic update of all copies of replicated scalar variables.
- You can export the distribution of an array to the visualizer to display it graphically.
- You see the HPF source and variables.
- You can set breakpoints in the HPF source code.

The following restrictions exist:

- You cannot display user-defined data types.
- Evaluation points and expressions are executed locally and cannot reference distributed arrays. However, you can use the **\$visualize** intrinsic. If you use the **\$visualize EVAL** intrinsic, remember that **EVAL** code is executed by every process. Therefore, you probably want to make this a non-shared action point.

In the address display for Variable Windows showing HPF variables, an additional field tells you whether the variable is distributed [**Dist**] or replicated [**Repl**]. If you update a replicated variable, it is updated in all processes. A distributed variable is only updated in its home process.



You cannot edit the address of a distributed array. If you edit the address of a replicated scalar, it will be marked as distributed, since it no longer makes sense to update all of the processes, as you do not know what is at that address in the other processes.

When you display an HPF distributed array, TotalView can also display the logical processor on which each element resides. You can change the display of this additional information for a single Variable Window by using the **View > Node Display** command. You can set the default for the TotalView session by using the `-hpf_node` or `-no_hpf_node` command-line options; you could also use the X resource `totalview*hpfNode` on page 279. No matter which way you set the default, you can always toggle the behavior in each window.

By default, this display is disabled. If it is enabled, TotalView displays a distributed array as is shown in Figure 60. Otherwise, the **Node** column is not displayed and a distributed array display looks the same as that of a normal array.

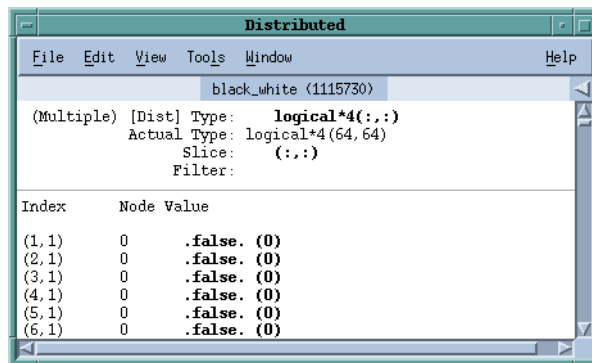


FIGURE 60: Block Distributed Array on Three Processes

## Starting TotalView with HPF

Beginning with PGI HPF release 2.4, TotalView can track processes started by `rpm` or `smp`, the default PGI HPF runtime libraries. If you still want to use MPI, you must ensure that the MPI implementation is supported by PGI HPF and TotalView. See “*Debugging MPI Applications Overview*” on page 77.

On IBM SP, or clusters of RS/6000 machines running IBM's parallel environment, you can use any runtime library started with the **poe** command.

On SGI IRIX, TotalView supports 64-bit PGI HPF programs only. You must compile your PGI HPF program with the **-64** compiler option.

### Dynamically Loaded Library

To debug PGI HPF code, TotalView needs to be able to dynamically load the file **libtvhpf.so**, which is distributed as part of the PGI HPF product.

TotalView searches for this file in the following order:

- 1 TotalView attempts to dynamically load the unadorned file name **libtvhpf.so**. This succeeds if **libtvhpf.so** is in one of the directories on your dynamic library path environment variable (on Sun Sparc SunOS5, IBM AIX, and SGI IRIX, this is **LD\_LIBRARY\_PATH** if the variable **LD\_LIBRARYN32\_PATH** is not set). On SGI IRIX, **libtvhpf.so** is in one of the directories on your **-n32** dynamic loader path (**LD\_LIBRARYN32\_PATH**).
- 2 If step 1 fails, TotalView uses the **PGI** environment variable to find the Portland Group installation. If the **PGI** environment variable is not set, TotalView looks for the default installation directory (**/usr/pgi**).
- 3 TotalView then searches the directories in the order shown in the following table.

TABLE 8: PGI HPF Dynamic Library Search Order

System	Search Path
IBM RS/6000 and SP AIX 4.x	\$PGI/sp2/lib
	\$PGI/rs6000/lib
Sun Sparc SunOS 5 (Solaris 2.x)	\$PGI/solaris/lib
SGI MIPS IRIX 6.x	\$PGI/sgi/lib-n32
	\$PGI/sgi/lib-64
	\$PGI/origin/lib/mips4

If TotalView still cannot locate a copy of **libtvhpf.so** and, if the TotalView verbosity level is not **silent**, an error message is displayed telling you that the library could not be found, and **HPF** debugging is disabled. TotalView will then start debugging the generated Fortran code.

If TotalView cannot find your copy of `libtvhpf.so`, you should either move it to one of the places that will be searched by default, or add its directory to your `LD_LIBRARY_PATH`.

## Setting Up PGI HPF Compiler Defaults

**NOTE** With PGI HPF version 2.4 and later, there is no need to use an MPICH-based runtime, and you can ignore this section.

Set up the HPF compiler with the defaults set for using MPICH, TotalView, IBM's parallel environment, and FORTRAN 77, as in the following sections.

If you have PGI HPF release 2.4, the `rc` files should already have been set up correctly, but they will use the default runtime, which is not MPI. If you want to use an MPI runtime, you should consult the PGI HPF manuals.

## Setting Up MPICH

You should follow the instructions in the PGI HPF manual and MPICH manual to ensure that you can build an HPF program and run it by using MPICH. One way to do this is to create your own `.pghpfrc` file and add lines similar to the following:

```
# Set up to use my MPI with PGI HPF
# Change the path to libmpi.a as appropriate
#
INCLUDE $DRIVER/.pghpfrc
set HPF_MPI=/where_your_mpi_lives/libmpi.a
set HPF_COMM_LIBS= \
    "-lpghpf_mpi$P $HPF_MPI $HPF_SOCKET"
```

Because these lines tell `pghpf` to use the MPI communications library, you do not need to name them on the command line at compilation time.

## Setting TotalView Defaults for HPF

To debug HPF code, you will normally set the breakpoint and barrier breakpoint behavior so that TotalView does not stop other processes when the breakpoint is hit. For more information, refer to "Parallel Debugging Tips" on page 117.

Other HPF resources are **totalview\*hpf** on page 278 and **totalview\*hpfNode** on page 279.

## Compiling HPF for Debugging

To compile your HPF program so it can be used with TotalView, you should use the **-g** and **-Mtotalview** options to **pghpf** when both compiling and linking. (The **-Mtv** option is the same as the **-Mtotalview** option.)

The **-g** option can produce confusing results when used by itself. For example, while you may see the HPF source code, none of the HPF debugging features work. If TotalView flags your HPF code in the stack backtrace as being **f77**, the program was probably not compiled with the **-Mtv** option.

If you want to debug the Fortran code generated by HPF, you must also use the **-Mkeepftn** option. Otherwise, the compiler deletes these intermediate Fortran files after it compiles the source code.

You can debug at the generated Fortran level by starting TotalView with the **-no\_hpf** option or setting the X resource **totalview\*hpf** to **false**. TotalView will then ignore the **.stb** and **.stx** files and show you the generated F77.

There is no need to relink the HPF program to debug at the generated Fortran level.

## Starting HPF Programs

The way in which TotalView starts an HPF parallel program depends on the machine on which the code is running and the runtime library linked into the HPF code.

### PGI HPF **smp** and **rpm** Libraries

Using TotalView to start a program linked with the **smp** and **rpm** libraries is similar to the way in which you would normally start the program. For example, suppose you start the program as follows:

```
my_program -bah -pghpf -np 6
```

Here is the command you would use to debug this file:

```
totalview my_program -a -bah -pghpf -np 6
```

## Starting HPF Programs with MPICH

In a workstation cluster environment using MPICH, debug your HPF application with TotalView by adding the `-tv` option to the `mpirun` command. For example, assume that you would begin executing your code with the following command:

```
mpirun -np 4 my_program
```

Using `mpirun`, you would invoke TotalView as follows:

```
mpirun -tv -np 4 my_program
```

## Workstation Clusters Using MPICH

Debugging workstation clusters uses the same mechanism as debugging an MPICH program since a compiled HPF program is an MPICH program. For more information, refer to "Debugging MPI Applications Overview" on page 77.

## IBM Parallel Environment

In the IBM parallel environment on an IBM SP or cluster of RS/6000 machines, parallel programs are started by using the `poe` command. Here is an example of starting TotalView on the `poe` command to debug a program named `hpf_test`:

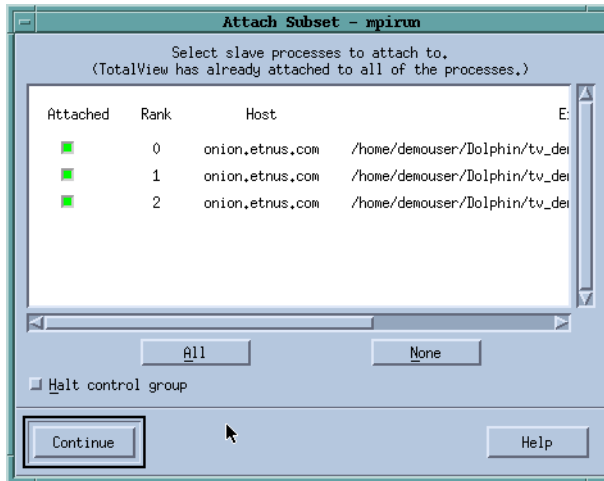
```
totalview poe -a hpf_test -procs 6
```

For more information, refer to "Starting TotalView on a PE Job" on page 86.

## Parallel Debugging Tips

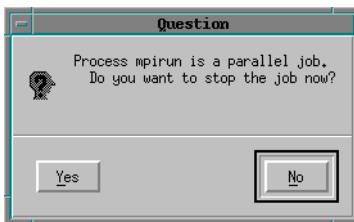
### Attaching to Processes

In a typical multiprocess job, you are interested in what is occurring in some of your processes and not as much interested in others. By default, TotalView tries to attach to all the processes that your program starts. If there are a lot of processes, there may be considerable overhead involved in opening and communicating with the jobs. You can minimize this overhead by using the **Group > Attach Subsets** command, which displays the following dialog box.

FIGURE 61: **Group > Attach Subset Dialog Box**

By selecting the boxes at the left of this dialog box, you tell TotalView which processes it should attach to. Restated, while your program will launch all of these processes, TotalView will only attach to the processes that you have selected here.

While you can use this command at any time, you would probably use it immediately before TotalView launches processes. Unless you have set preferences otherwise, TotalView will stop and ask if you want it stop your processes. (See Figure 62.)

FIGURE 62: **Stop Before Going Parallel Question Dialog Box**

This is a good time to use this command.

The commands on the **Parallel** Page with the **File > Preferences** dialog box let you control what TotalView will do when your program goes parallel. Here is the Parallel page:

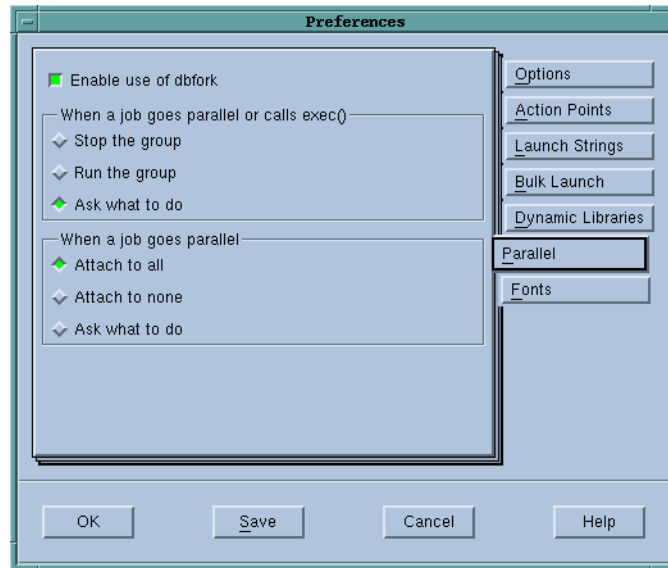


FIGURE 63: **File > Preferences: Parallel Page**

The **When a job goes parallel or calls exec()** radio buttons have the following meanings:

- **Stop the group:** Stop the control group immediately after the processes are created.
- **Run the group:** Allows all newly created processes in the control group to run freely.
- **Ask what to do:** Asks what should occur. If you select this option, TotalView will ask if it should start the created processes.

The **When a job goes parallel** radio buttons have the following meaning:

- **Attach to all:** TotalView automatically attaches to all processes when they begin executing.
- **Attach to none:** TotalView will not attach to any created process when it begins executing.

- **Ask what to do:** Asks what should occur. If you select this option, TotalView opens the same dialog box that is displayed when you select **Group > Attach Subsets**. TotalView will then attach to the processes that you have selected. Note that this dialog box is not displayed when you set the preference. Instead, it controls what will happen when your program creates parallel processes.

## General Parallel Debugging Tips

Here are some tips that are useful for debugging most parallel programs:

### ■ Breakpoint behavior

When you are debugging message-passing and other multiprocess programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multiprocess program hits a breakpoint, TotalView will stop all the other processes.

To change the default stopping action of breakpoints and barrier breakpoints, you can set TotalView preferences. Information on this preferences can be found in the online help.

A second method is to specify the `-no_stop_all` TotalView command-line options described on page 300 and `-no_barr_stop_all` described on page 291.

These settings set breakpoint and barrier breakpoint behavior. These options tell TotalView if it should allow other processes and threads to continue to run when a process or thread hits the breakpoint.

These options only affect the default behavior. As usual, you can choose a behavior for a breakpoint by setting the breakpoint properties in the **File > Preference's Action Points** Pane. See "*Breakpoints for Multiple Processes*" on page 209.

### ■ Process synchronization

TotalView has two features that make it easier to get all of the processes in a multiprocess program synchronized and executing a line of code.

Process barrier breakpoints and the process hold/release features work together to help you get control the execution of your processes. See "*Barrier Breakpoints*" on page 212.

The Process Window's **Group > Run To** command is a special kind of stepping command. It allows you to run a group of processes to a selected source line or instruction. See "*Group-Width Stepping*" on page 141.



## ■ Using group commands

Group commands are often more useful than process commands.

It is often more useful to use the **Group > Go** command to restart the whole application instead of the **Process > Go** command. You would then use the **Group > Halt** command instead of **Process > Halt**.

The group-level single-stepping commands such as **Group > Step** and **Group > Next** allow you to single-step a group of processes in a parallel. See “Group-Width Stepping” on page 141.

## ■ Process-level stepping

If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to be hung (it continuously displays the watch cursor). If you single-step a process over a statement that cannot complete without allowing another process to run and that process is stopped, the stepping process appears to hang. This can occur, for example, when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by selecting **Cancel** in the **Waiting for Command to Complete** window that will appear. As an alternative, consider using a group-level single-step command instead.

**NOTE** Etnus receives many bug reports about processes being hung. In almost all cases, the reason is that one process is waiting for another. Using the Group debugging commands almost always solves this problem.

## ■ Determining which processes and threads are executing

The TotalView Root Window helps you determine where various processes and threads are executing. When you select a line of code in the Process Window, the Root Window Attached Page is updated to show which processes and threads are executing that line. See “Displaying Thread and Process Locations” on page 147.

## ■ Viewing variable values

You can view (lamine) the value of a variable that is replicated across multiple processes or multiple threads in a single Variable Window. See “Displaying a Variable in All Processes or Threads” on page 196.

## ■ Restarting from within TotalView

You can restart a parallel program at any time. If your program runs too far, you can kill the program by selecting the **Group > Delete** command.

This command kills the master process and all the slave processes. Restarting the master process (for example, **mpirun** or **poe**) recreates all of the slave processes. Startup is faster when you do this because TotalView does not need to reread the symbol tables or restart its server processes since they are already running.

## MPICH Debugging Tips

Here are some debugging tips that apply only to MPICH:

### ■ Passing options to mpirun

You can pass options to TotalView through the MPICH **mpirun** command.

To pass options to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **-no\_stop\_all** option as in the following C shell, example:

```
setenv TOTALVIEW "totalview -no_stop_all"
```

### ■ Using ch\_p4

If you start remote processes with MPICH/**ch\_p4**, you may need to change the way TotalView starts its servers.

By default, TotalView uses **rsh** to start its remote server processes. This is the same behavior as **ch\_p4**. If you configure MPICH/**ch\_p4** to use a different start-up mechanism from another process, you will probably also need to change the way that TotalView starts the servers.

For more information about **tvdsrv** and **rsh**, see "Single Process Server Launch Options" on page 62. For more information about **rsh**, see "Single Process Server Launch Command" on page 66.

## IBM PE Debugging Tips

Here are some debugging tips that apply only to IBM MPI (PE):

### ■ Avoid unwanted timeouts

You can cause undesired timeouts if you place breakpoints that stop other process too soon after calling **MPI\_Init()** or **MPL\_Init()**. If you create "stop all" breakpoints, the first process that gets to the breakpoint stops all the other parallel processes that have not yet arrived at the breakpoint. This may cause a timeout.

To turn the option off, select the Process Window's **Action Point > Properties** command while the line with the stop symbol is selected. After the **Properties** dialog box appears, you should deselect the **Plant in share group** check box.

#### ■ **Control the poe process**

Even though the **poe** process continues under TotalView control, you should not attempt to start, stop, or otherwise interact with it. Your parallel tasks require that **poe** continue to run. For this reason, if **poe** is stopped, TotalView automatically continues it when you continue any parallel task.

#### ■ **Avoid slow processes due to node saturation**

If you try to debug a PE program in which more than three parallel tasks run on a single node, the parallel tasks on each node may run noticeably slower than they would run if you were not debugging them.

This becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks may make hardly any progress. This is because PE uses the **SIGALRM** signal to implement communications operations, and AIX requires that debuggers must intercept all signals. As the number of parallel tasks on a node increases, TotalView becomes saturated, and cannot keep up with the **SIGALRM**s being sent, thus slowing down the tasks.





## Chapter 6

# Debugging Programs

This chapter explains how to perform basic debugging tasks with TotalView. The topics discussed are:

- Displaying Your Program's Call Tree
- Find Code As You Are Debugging
- Display Your Code in Source and Assembler Formats
- Return to the Currently Executing Line in the Stack Frame
- Invoke Your Editor on Source Files You Are Debugging
- Interpret Status and Control Registers
- Use Commands for Controlling Processes and Threads
- Control Process Groups in Multiprocess Programs
- Set Action Points
- Use Single-step Commands
- Set The Program Counter

## Displaying Your Program's Call Tree



Debugging is an art, not a science. Locating a problem is often 90% or more of the effort. Debugging often means having the "intuition" to know what a problem means and where to look for it. TotalView's call tree is one tool that helps you get an understanding of what your program is doing so that you can begin to understand how your program is executing.

Use the **Tools > Call Tree** command in the Process Window to tell TotalView to display a call tree window. (See Figure 64.)

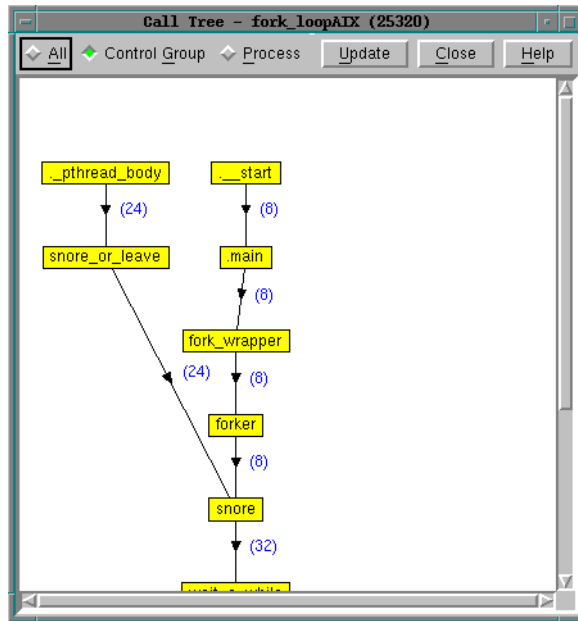


FIGURE 64: **Tools > Call Tree** Dialog Box

The call tree is a diagram showing all the currently active routines. These routines are linked by arrows indicating that one routine is called by another. This call tree is a *dynamic* call tree in that it displays the call tree at the time when TotalView creates the call tree. The **Update** button tells TotalView to recreate this display.

You can tell TotalView to display a call tree for all processes and threads, just the current process, or for this process's control group. If the call tree is for a multiprocess or multithreaded program, numbers next to the arrows indicate how many times a routine was called when the tree was created.

As you begin to understand your program, you will see that your program may have a rhythm and a dynamic that is reflected in this diagram. As you examine this structure, you will sometimes see things that do not look

right—which is a subjective response to the data. These places are often where you want to begin looking for problems.

Looking at the call tree can also tell you where bottlenecks are occurring. For example, if one routine is used by many other routines and that routine controls a shared resource, this thread may be negatively affecting performance. For example, in the previous figure the **snore** routine might be a bottleneck.

## Finding the Source Code for Functions

You can search for a function's declaration by selecting the **View > Lookup Function** command and typing a function name within the following dialog box.

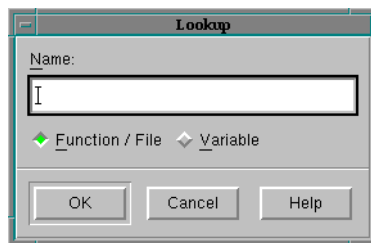


FIGURE 65: **View > Lookup Function** Dialog Box

After TotalView finds the source code, it displays it in the Source Pane. If the function you selected was not compiled with source-line information, TotalView displays disassembled machine code.

**NOTE** When you want to return to the previous contents of the Source Pane, use the undive icon located in the upper right corner of the Source Pane and just below the Stack Frame Pane. You can also use the **View > Reset** command to discard the dive stack so that the Source Pane is displaying the PC it displayed when it was first stopped.

You can use the **File > Edit Source** command (see “Editing Source Text” on page 132 for details) or an X Window System client such as **xmore**, **vi**, or **emacs** to display these files.

Another method of locating a function's source code is by diving into it from within the Source Pane.

## Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you may have specified the name of:

- A static function and your program contains multiple static functions by that same name.
- A member function in a C++ program and there are multiple classes with member functions of that name.
- An overloaded function or a template function.

Figure 66 shows an example of the dialog that TotalView displays when it encounters an ambiguous function name.

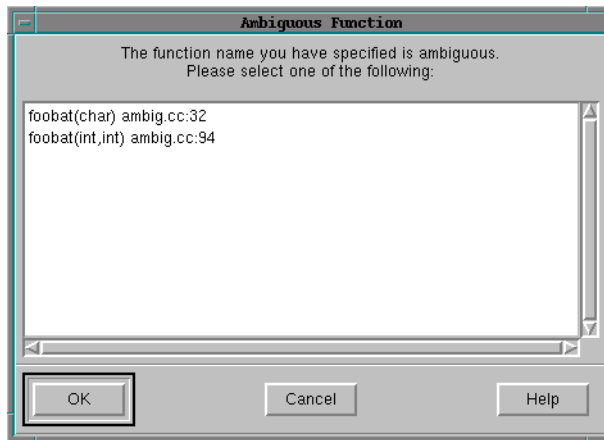


FIGURE 66: **Ambiguous Function Dialog Box**

To resolve the ambiguity, click one of the function names.

TotalView may request that you select a function when you:

- Specify a function name with the **View > Lookup Function** command.
- Dive on a name in the Source Pane.



- Set a breakpoint at a line in a function.
- Select a function by clicking on its name in the Stack Trace Pane.

## Finding the Source Code for Files

You can display a file's source code by selecting the **View > Lookup Function** command and entering the file name in the dialog box shown in Figure 67.

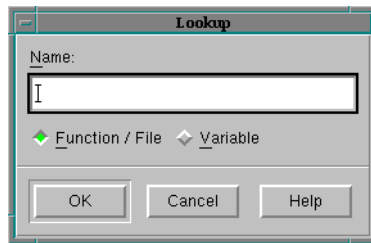


FIGURE 67: **View > Lookup Function** Dialog Box

You can enter the name of a header file if the header file contains source lines that produce executable code.

## Examining Source and Assembler Code

You can display your program in several different ways. If you display assembler in the Source Pane, you can also display addresses in the following ways:

### Source code (Default)

Use the **View > Source As > Source** command.

**Assembler code** Use the **View > Source As > Assembler** command.

### Source and assembler interleaved

Use the **View > Source As > Interleaved** command.

Source statements are treated as comments. You can set breakpoints or evaluation points only at the machine level. Setting an action point at the first instruction after a source statement, however, is equivalent to setting it at that source statement.

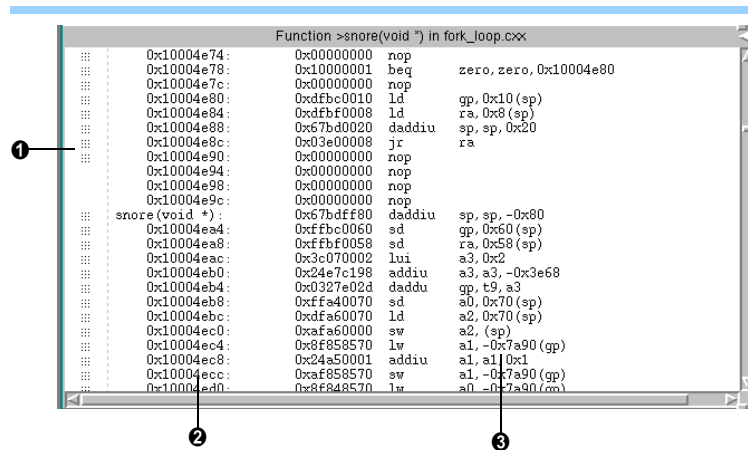
You can tell TotalView to display assembler code using symbolic or absolute addresses:

TABLE 9: Assembler Code Display Styles

To Display Address Using	Use This Command
Absolute addresses for locations and references—default	View > Assembler > By Address
Symbolic addresses (function names and offsets) for locations and references	View > Assembler > Symbolically

The following three figures illustrate the different ways TotalView can display assembler code.

**NOTE** You can also display assembler instructions in a Variable Window. For more information, see “Displaying Machine Instructions” on page 157.



- ① Gridset (dotted grid) indicates action point can be set on an instruction
- ② Location by absolute address
- ③ References by absolute address

FIGURE 68: Address Only (Absolute Addresses)

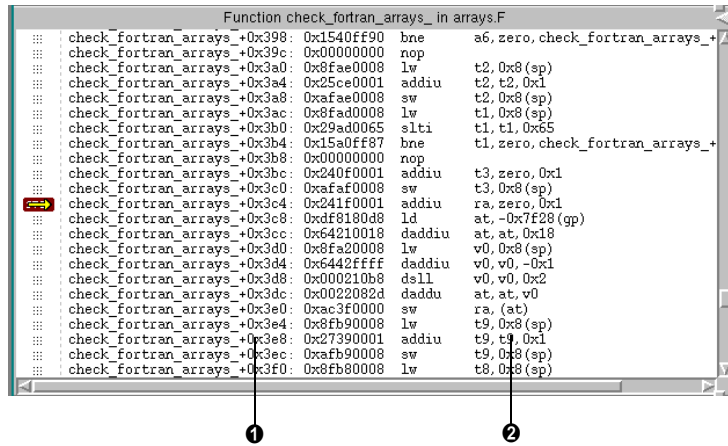


FIGURE 69: Assembler Only (Symbolic Addresses)

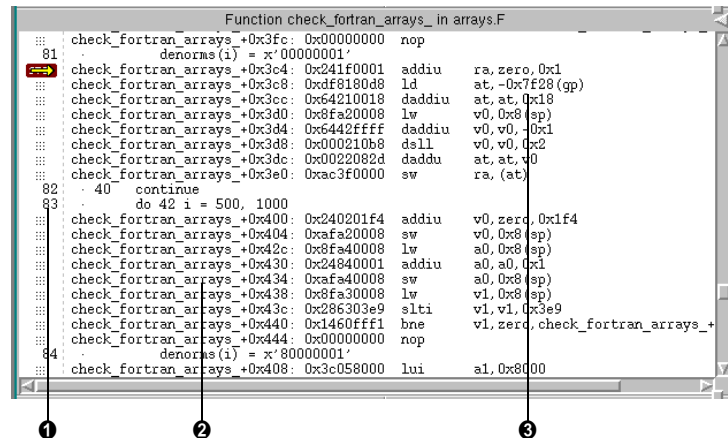


FIGURE 70: Interleaved Source/Assembler (Absolute Addresses)

## Resetting the Current Stack Frame

You can return to the executing line of code for the current stack frame by selecting the **View > Reset** command. This command forces the PC arrow onto the screen and discards the dive stack.

This command is also useful if you want to undo the effect of scrolling or of moving to different locations using, for example, the **View > Lookup Function** command. For details, see "*Finding the Source Code for Functions*" on page 127.

If the program has not begun to run, the **View > Reset** command displays the first executable line in your main program function or subroutine.

## Editing Source Text

You can use the **File > Edit Source** command to edit source files while you are debugging. TotalView starts your editor on the source file being displayed in the Source Pane of the Process Window.

TotalView uses an *editor launch string* to determine how to start your editor. TotalView expands this string into a command that is then executed by the **sh** shell.

The fields within **File > Preferences's Launch Strings** page let you change the editor and the way TotalView launches the editor. The Help for this page contains information on how you set this preference.

You can also change the editor launch string by using a TotalView preference. You can find information on this in the online help.

## Using the Toolbar to Select a Target

The Process Window's toolbar can be divided into three parts. The first part defines the scope of the command selected in the second part of the toolbar. (The third part, which is not shown, changes the Process Window's display between processes and threads.) A few examples will make this clear. Figure 71 shows the left portion of the toolbar.

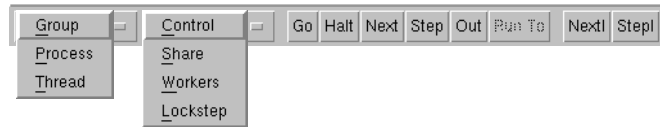


FIGURE 71: The Toolbar

**NOTE** See Chapter 3 of the CLI for a discussion of groups, processes, and threads.

The first pulldown has three elements on it: **Group**, **Process**, and **Thread**. Your choices here indicate the width of the command. For example, if **Group** is selected, a **Go** command will continue the group. Which group TotalView will continue is set by the choices on the second pulldown. If, for example, **Group** and **Control** are selected, then **Go** continues the control group, **Step** single-steps the control group, and so on.

In most cases, you will choose items within the first two elements, then manipulate your program by using the remaining command buttons. In most cases, you will use the following two combinations:



FIGURE 72: Toolbar Combinations

For example, most MPI programs are debugged using the bottom setting.

## Stopping Processes and Threads

To stop a group, process, or thread, go to the Process Window and select a **Halt** command from the **Group**, **Process**, or **Thread** pulldowns:

The **Halt** commands differ in the scope of what they halt. In all cases, TotalView uses the current thread (which is called the thread of interest or TOI) to determine what else it will halt. For example, **Process > Halt** stops the process that contains the TOI. When you use **Group > Share > Halt**, TotalView determines what other threads are in the same share group as the TOI, then stops the processes associated with these threads.

After entering a **Halt** command, TotalView updates the Process Window and all related windows. When you restart the process, execution continues from the point where the process stopped.

## Updating Process Information

You can force the Process Window to update process information by using the **Window > Update** command. This command tells TotalView that it should temporarily stop the process so that it can reread the thread registers and memory. After this information is acquired, the process continues executing. This allows you to quickly refresh your view of a process.

## Holding and Releasing Processes and Threads

TotalView allows you to hold and release processes and threads. When something is held, all commands that tell it to run, such as **Process > Go** or **Group > Go**, have no effect.

Manually holding and releasing processes and threads is useful if:

- You wish to run a subset of the processes and threads, you can manually hold all but the ones you want to run.
- A process is held at a barrier point and you want to run it without first running all the other processes in the group to that barrier, you can release it manually and then run it. Thread behavior at thread barriers is similar.

A process or thread may also be held if it stops at a barrier breakpoint. You can manually release a process or thread being held at a barrier breakpoint. See “*Barrier Breakpoints*” on page 212 for more information on manually holding and releasing barrier breakpoint.

When a process is being held, the Root Window and Process Window display a held indicator. (This is a letter **H**.) When a thread is being held, the letter displayed is **h**.

Here are the ways to hold or release a thread, process, or group of processes:

- You can hold a group of processes by choosing the **Group > Hold** command.
- You can then release the group of processes by choosing the **Group > Release** command.
- You can toggle the hold/release state of a process by selecting and deselecting the **Process > Hold** command.
- You can toggle the hold/release state of a thread by selecting and deselecting the **Thread > Hold** command.

If a process or a thread is running when you issue a hold or release command, TotalView first stops the process or thread then holds it.

**NOTE** Releasing a process does not mean that the thread will resume executing; execution only resumes after you use one of the execution commands. In addition, TotalView allows you to hold and release processes independently from threads. That is, changing a process's hold state does not affect its threads' hold state and vice versa.

Notice that the Process pulldown also contains a **Hold Threads** and **Release Threads** command. If you select **Hold Threads**, the scope of what is held is the same as when you select **Hold**. This command, however, is used for another purpose. Assume that you are debugging a process with fifty threads and you want only a few of them to run. You could select **Hold Threads**, then go to the **Threads** menu to release only those that you want to run. That is, this command and **Release Threads** are convenience functions that can save you some work.

## Examining Groups

When you debug a multiprocess program, TotalView adds each process to a control group and a share group as the process starts.

**NOTE** These groups are not related to UNIX process groups or PVM groups.

TotalView groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. The two types of process groups are:

- Control Group** Includes the parent process and all related processes. A control group includes children that a process forked (processes that share the same source code as the parent). It also included forked children that subsequently call a function such as **execve()**. That is, a control group can contain processes that do not share the same source code as the parent.
- Control groups also include processes created in parallel programming disciplines like MPI.
- Members of a control group can be stopped as a group.
- Share Group** Is the set of processes within a control group that share the same source code. Members of the same share group share action points.

**NOTE** A full discussion of Groups, Processes, and Threads can be found in Chapter 3 of the CLI Guide. This information is also contained within Help. In addition, you can download this information from our Web site.

TotalView automatically creates share groups when your processes fork children that call **execve()** or when processes using the same code are created in some parallel programming models such as MPI.

TotalView names processes based upon the name of the source program. Here are the naming rules TotalView uses:

- TotalView names the parent process after the source program.
- Child processes that are forked have the same name as the parent, but with a numerical suffix (*.n*). If you are running an MPI program, the numeric suffix is the process's rank in **COMM\_WORLD**.
- Child processes that call **execve()** after they are forked have the parent's name, the name of the new executable in angle brackets (<>) and a numerical suffix.

For example, if the **generate** process does not fork any children, and the **filter** process forks a child process that subsequently calls itself and then calls **execve()** to execute the **expr** program, TotalView names and groups the processes as shown in Figure 73.



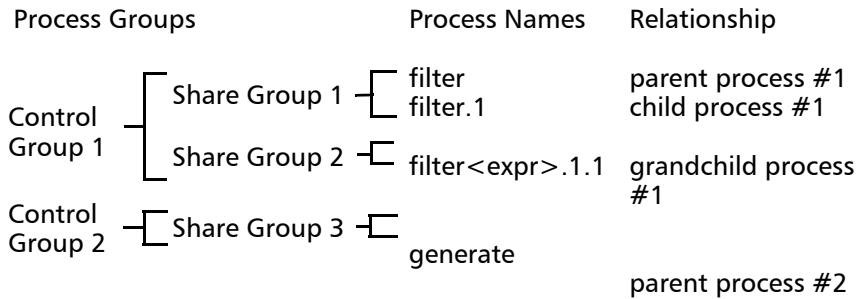
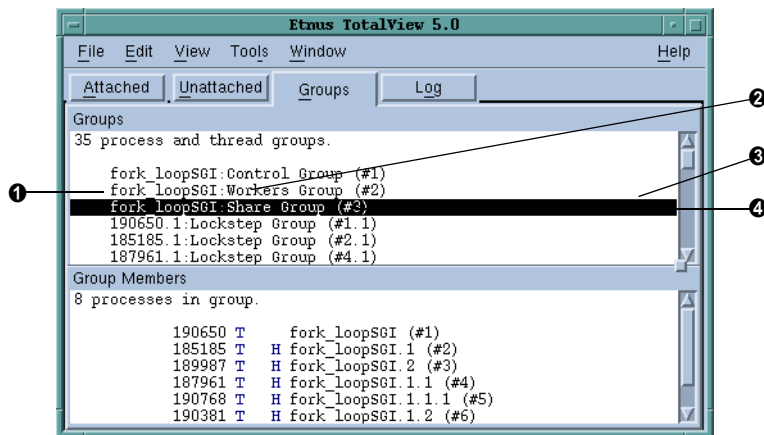


FIGURE 73: Example of Control Groups and Share Groups

## Displaying Groups

The Root Window's Attached Page displays the names of individual processes. To display a list of process and thread groups, select the Root Window's Group Tab. See Figure 74.



- ❶ Name of executable
- ❷ Type of process or threadgroup
- ❸ Select a group in the top pane to display members

FIGURE 74: Root Window: Groups Page

When you select a group in the top list, TotalView updates the bottom list to show the group's members. After the bottom portion is updated, you can dive into any of these processes or threads by double-clicking on it.

## Placing Processes into Groups

TotalView uses your executable's name to determine the share group to which the program belongs. It does not examine the program to see if it is identical to another program with the same name; TotalView assumes the programs are identical because their names are identical.

TotalView does not expand a program's full path name, so if one instance of a program is named with the full path name (`./foo`), and another is named with the file name (`foo`), the programs are placed in different share groups.

## Starting Processes and Threads

To start a process, go to the Process Window and select a **Go** command from the **Group**, **Process**, or **Thread** pulldowns.

After you enter a **Go** command, TotalView decides what it will run based on the current thread. It uses this thread to decide what other threads it should run. For example, if you enter **Group > Workers > Go**, TotalView continues all threads in the workers group associated with this thread.

The most often used commands are **Group > Go** and **Process > Go**. The **Group > Go** command creates and starts this process and all other processes in the multiprocess program (control group). It resumes execution of this process and the execution of all processes in the control group if the process:

- Is not being held
- Already exists and is stopped, or
- Is at a breakpoint.

Issuing **Group > Go** on a process that's already running starts the other members of the control group.

**Process > Go** creates and starts this process. It resumes execution if the process is not being held, already exists and is stopped, or is at a break-point. Starting a process causes all threads in the process to resume execution unless the thread is held.

**NOTE** **Thread > Go** is disabled if asynchronous thread control is not available.

For a single-process program, **Process > Go** and **Group > Go** are equivalent. For a single-threaded process, **Process > Go** and **Thread > Go** are equivalent.

**NOTE** If TotalView is holding a process or thread, these commands will not start the process or threads. See *"Holding and Releasing Processes and Threads"* on page 134.

## Creating a Process Without Starting It

The **Process > Create** command creates a process and stops it before any of your program executes. If a program is linked with shared libraries, TotalView allows the dynamic loader to map into these libraries. Creating a process without starting it is useful if you need to:

- Create watchpoints or change global variables after a process is created, but before it runs.
- Debug C++ static constructor code.

## Creating a Process by Single-Stepping

The TotalView single-stepping commands allow you to create a process and run it to a location in your programs. The single-stepping commands in the **Process** menu are as shown in the following table.

TABLE 10: Creating a Process by Stepping

Command	Creates the process and ...
<b>Process &gt; Step</b>	Runs it to the first line of the <b>main()</b> routine.
<b>Process &gt; Next</b>	Runs it to the first line of the <b>main()</b> routine; this is the same as <b>Process &gt; Step</b> .
<b>Process &gt; Step Instruction</b>	Stops it before any of your program executes.

TABLE 10: Creating a Process by Stepping (cont.)

Command	Creates the process and ...
<b>Process &gt; Next Instruction</b>	Runs it to the first line of the <b>main()</b> routine. this is the same as <b>Process &gt; Step</b> .
<b>Process &gt; Run To</b>	Runs it to the line or instruction selected in the Process Window.

If a group-level or thread-level stepping command creates a process, the behavior is the same as if it were a process-level command.

## Stepping

TotalView's stepping commands allow you to:

- Execute one source line or machine instruction at a time.
- Run to a selected line, which acts like a temporary breakpoint.
- Run until a function call returns.

Single-step commands are on the **Group**, **Process**, and **Thread** menus, and operate at group, process, or thread width. This width affects which threads within a process and processes within a group TotalView allows to run while the single-stepping command is executing.

In all cases, stepping commands operate on the TOI, which is the selected thread in the current Process Window.

On all platforms except SPARC Solaris, TotalView uses *smart* single stepping to speed up stepping of one-line statements containing loops and conditions, such as Fortran 90 array assignment statements. *Smart* stepping occurs when TotalView realizes that it does not need to step through an instruction. For example, assume that you have the following statements:

```
integer iarray (1000,1000,1000)
iarray = 0
```

These two statements cause one billion scalar assignments. If your machine steps every instruction, you will probably never get past this statement. *Smart* stepping means that TotalView will single step through the assignment statement at a speed that is very close to your machine's native speed.

## Process-Width Stepping

The behavior of process-width stepping commands depends on whether the Group of Interest (GOI) is set to a process group or a thread group.

### GOI is a process group

TotalView runs the process until the Thread of Interest (TOI) arrives at its goal location, which can be the next statement, the next instruction, and so forth. When it reaches the goal, TotalView stops the process and the command completes.

### When the GOI is a thread group

The behavior differs. All threads in the GOI and all manager threads are allowed to run. As each member of the GOI arrives at the goal, it is stopped; the rest of the threads are allowed to continue. The command ends when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

**NOTE** The *Run To* commands are similar, but there are some important differences. See “*Executing to a Selected Line*” on page 144 for more information.

## Group-Width Stepping

The behavior of group-width stepping commands depends on whether the Group of Interest (GOI) is a process group or a thread group.

### GOI is a process group

TotalView examines the group and identifies each process in it having a thread stopped at the same location as the TOI (a *matching* process). TotalView runs all processes in the control group associated with the process of interest (POI). Each time a thread arrives at the goal, the process containing that thread is stopped. The command finishes when TotalView stops all “matching” processes. At that time, all members of the control group are also stopped.

### GOI is a thread group

TotalView also runs all processes in the control group. However, as a thread arrives at the goal, just that thread is stopped; the rest of the threads in the pro-

cess containing it are allowed to continue. The command finishes when all threads in the GOI have arrived at the goal. (Threads that are not in the same share group as the TOI are not waited for, since they are executing different code, and can never arrive at the goal.) When the command finishes, all processes in the control group are again stopped.

**NOTE** The *Run To* commands are similar, but there are some important differences. See “*Executing to a Selected Line*” on page 144 for more information.

## Thread-Width Stepping

When TotalView executes a thread-width stepping command, it steps the TOI while holding the other threads in the process.

**NOTE** Thread stepping is not implemented on Sun platforms. On SGI platforms, thread stepping is not available with pthread programs. If, however, your program is based on SGI compiler-generated parallelism such as used in OpenMP, thread stepping is available.

Stepping a thread tells TotalView that it should just run that thread. TotalView also allows all manager threads to run freely while it steps the TOI.

Thread-level single-step operations can fail to complete if the TOI needs to synchronize with a thread that is not running. For example, if the TOI requires a lock that another held thread owns, and steps over a call that tries to acquire the lock, the primary thread cannot continue successfully. You must allow the other thread to run in order to release the lock. If this applies, you should instead use process width.

## Selecting Source Lines

Several of the single-stepping commands require you to select a source line or machine instruction in the Source Pane. To choose a source line, place the cursor over the line and select it. See “*Displaying Thread and Process Locations*” on page 147 for information on what occurs within the Root Window when you select a source line or machine instruction.

If you select a source line that has more than one instantiation (for example, in a C++ function template or code in a header file), TotalView displays its Ambiguous Line dialog box that allows you to select a specific instantiation, as shown in the following figure.

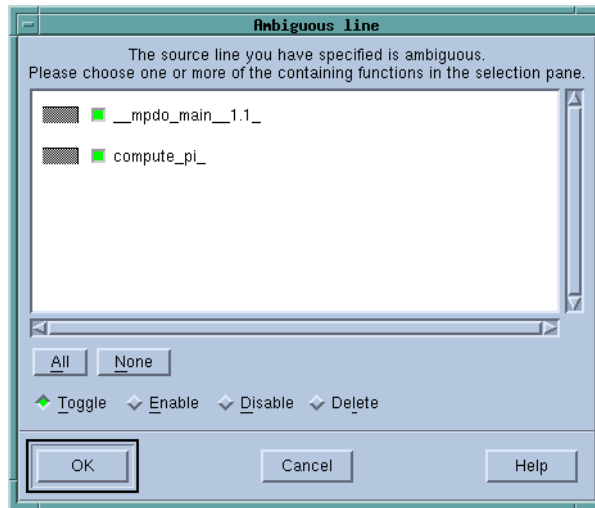


FIGURE 75: **Ambiguous Line Dialog Box**

You can now select the function in which the line is located so TotalView can determine where it should place its goal.

## Using Single-Step Commands

While different programs have different requirements, the most common debugging stepping mode is to set group focus to **Control** and the target to **Process** or **Group**. You can now select stepping commands from the **Process** or **Group** menus or use commands in the icon bar.

The following applies to all single-step command:

- To cancel a single-step command, put the mouse pointer in the Process Window and select **Group > Halt** or **Process > Halt**.

- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you issue a source-line step command and the primary thread is executing in a function that has no source-line information, TotalView performs the corresponding instruction step instead.

When TotalView steps, it steps a line at a time. This means that if you have more than one statement on a line a step instruction executes all the instructions.

## Stepping into Function Calls

The stepping functions execute a single source line or instruction. If the source line or instruction names a function, TotalView steps into it. If the source does not exist, TotalView displays the machine instructions for the function.

TotalView has eight **Step** commands and eight **Step Instruction** commands. These commands are located on the **Group**, **Process**, and **Thread** pulldowns.

## Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call.

TotalView has eight **Next** commands that execute a single source line while stepping over functions and eight **Next Instruction** commands that execute a single machine instruction while stepping over functions. These commands are located on the **Group**, **Process**, and **Thread** pulldowns.

## Executing to a Selected Line

You do not have to set a breakpoint to stop execution at a specific line because TotalView lets you run your program to a selected line or machine instruction. After selecting the line on which you want the program to



stop, invoke one of the eight **Run To** commands. These commands are located on the **Group**, **Process**, and **Thread** pulldowns.

The **Run To** commands do not work like the other group single-step commands. Here is what you should know if you are running at process width:

- Process group** If the TOI is already at the goal location, TotalView steps the TOI past the line before the process is run. This allows you to use the **Run To** command repeatedly within loops.
- Thread group** If any thread in the process is already at the goal. It is temporarily held while other threads in the process run. After all threads in the thread group reach the goal, TotalView stops the process. This allows you to synchronize the threads within the process of interest at a source line.

If you are running at group width:

- Process group** TotalView examines each process in the process and share group to determine if at least one thread is already at the goal. If a thread is at the goal, TotalView holds its process. Other process are allowed to run. When at least one thread from each of these processes is held, the command completes. This lets you synchronize at least one thread in each of these processes at a source line. If you are running a control group, this synchronizes all processes in the share group.
- Thread group** TotalView examines all the threads in the thread group that are in the same share group as the TOI to determine if a thread is already at the goal. If it is, TotalView holds it. Other threads are allowed to run. When all of the threads in the TOI's share group reach the goal, TotalView stops the TOI's *control* group and the command completes. This lets you synchronize thread group members. If you are running a workers group, this synchronizes all worker threads in the share group.

The process stops when the TOI and at least one thread from each process in the group or process reach the command stopping point. This lets you synchronize a group of processes and bring them to one location.

You can also run to a selected line in a nested stack frame, as follows:

- 1 Select a nested frame in the Stack Trace Pane.
- 2 Select a source line or instruction within the function.
- 3 Issue a **Run To** command.

TotalView executes the primary thread until it reaches the selected line in the selected stack frame.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP to the value of the current FP:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView automatically continues your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

If your program reaches a breakpoint while running to a selected line, the debugger discards the “run to” operation and stops at the breakpoint.

## Executing to the Completion of a Function

You can step your program out of a function call. To finish executing the current function in a thread, select one of the eight **Out** commands. These commands are located on the **Group**, **Process**, and **Thread** pull-downs.

If the source line that is the *goal* of the **Out** operation has more than one statement, TotalView will stop you just after the routine from which just emerged. This allows you to step into the next routine on the line.

When one of these command completes, the primary thread is left stopped at the instruction after the one that called the function.

You can also return out of several functions at once, by selecting a nested stack frame in the Stack Trace Pane and then issuing an **Out** command.

TotalView executes the primary thread until it returns to the function in the selected frame.

If your program calls recursive functions or mutually recursive functions, you can select a nested stack frame in the Stack Trace Pane to tailor completion of the function even more. In this situation, TotalView uses the frame pointer (FP) of the selected stack frame and the selected source line or instruction to determine when to stop execution. When your program reaches the selected line, TotalView compares the value of the selected FP with the value of the current FP:

- If the value of the current FP is deeper (more deeply nested) than the value of the selected FP, TotalView continues executing your program.
- If the value of the current FP is equal or shallower (less deeply nested) than the value of the selected FP, TotalView stops your program.

## Displaying Thread and Process Locations

You can see which processes and threads in the share group are at a location by selecting a source line or machine instruction in the Source Pane of the Process Window. TotalView dims thread and process information in the Root Window's Attached Page for share group members if the thread or process is not at the selected line. A process is considered to be at the selected line if any of the threads in the process are at that line. Selecting a line in the Process Window that is already selected removes the dimming in the Attached Page.

The Attached Page reflects the line that you selected most recently. If you have several Process Windows open, the display in the Attached Page will change depending on the line you selected last in a Process Window. The display can also change after an operation that changes the process state or when you issue a **Window > Update** command.

Figure 76 shows an Attached Page with dimmed process information. In this example, the parallel program was run to a barrier breakpoint, and one process (`mpirun<cpi>.0`) was single-stepped to the next source line.

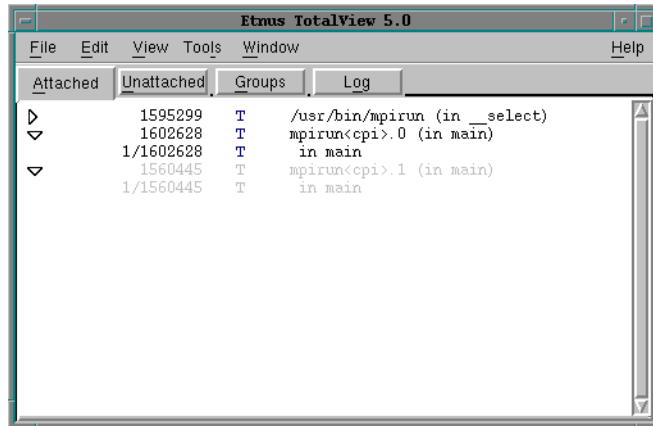


FIGURE 76: Dimmed Process Information in the Root Window

Note that since the MPI starter process (`mpirun`) is not in the same share group as the processes running the `cpi` program, the process information is not dimmed.

## Continuing with a Specific Signal

Letting your program continue to execute with a specific signal is useful when your program contains a signal handler. Here is how you tell TotalView that this should occur:

- 1 Select the Process Window's **Thread > Continuation Signal** command.
- 2 In the displayed dialog box, type the name select the signal to be sent to the thread.
- 3 Select **OK**.

The continuation signal is set for the thread you are focused on in the Process Window. If the operating system can deliver multithreaded signals, you may set a separate continuation signal for each thread. If it

cannot, this command clears continuation signals set for other threads in the process.

- 4 Continue execution of your program with commands such as **Process > Go**, **Step**, **Next**, or **Detach**.

TotalView continues the threads with the specified signals.

**NOTE** You can clear the continuation signal by selecting signal 0.

## Setting the Program Counter

You might find it useful to resume the execution of a thread at some statement other than the one where it stopped. You can do this by resetting the value of the program counter (PC). For example, you might want to skip over some code, execute some code again after changing certain variables, or restart a thread that is in an error state.

Setting the PC can be crucial when you want to restart a thread that is in an error state. Although the PC icon in the tag field points to the source statement that caused the error, the PC actually points to the failed machine instruction within the source statement. You need to explicitly reset the PC to the correct instruction. (You can verify the actual location of the PC before and after resetting it by displaying it in the Stack Frame Pane or displaying interleaved source and assembler code in the Source Pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line, a selected instruction, or an absolute value (in hexadecimal). When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is all you need to do.

To set the PC to a selected line:

- 1 If you need to set the PC to a location somewhere within a line of source code, display the **View > Source As > Interleaved** command. TotalView responds by displaying the assembler code.
- 2 Select the source line or instruction in the Source Pane. TotalView highlights.

If you select a line in a C++ function template that has more than one instantiation, TotalView asks you to select an instantiation. See “*Selecting Source Lines*” on page 142 for a description of how this works.

- 3 Select the **Thread > Set PC** command. TotalView asks for confirmation, resets the PC, and moves the PC icon to the selected line.

When you select a line and ask TotalView to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement in the currently selected stack frame. If the currently selected stack frame is not the top stack frame, TotalView asks if it can unwind the stack:

This frame is buried. Should we attempt to unwind the stack?

If you select **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame. If you select **No**, TotalView sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you cannot assume that the stack and registers have correct values, selecting **No** is not usually the right thing to do.

## Deleting Programs

To delete all the processes in a control group, select the **Group > Delete** command. The next time you start the program, for example, by using the **Process > Go** command, TotalView creates and starts a fresh master process.

## Restarting Programs

You can use the **Group > Restart** command to restart a program that is running or one that is stopped but has not exited.

If the process is part of a multiprocess program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

The **Group > Restart** command is equivalent to the **Group > Delete** command followed by the **Process > Go** command.

## Checkpointing Programs and Processes

On SGI IRIX platforms, you can save the state of selected processes and then use this saved information to restart the processes from the position where they were saved. For more information, see the Process Window's **Tools > Create Checkpoint** and **Tools > Restart Checkpoint** commands in TotalView's Help information. In addition, you can also perform these same activities from within the CLI by using the **dcheckpoint** and **drestart** commands.

## Interpreting Status and Control Registers

The Stack Frame Pane in the Process Window lists the contents of CPU registers for the selected frame—you may need to scroll past the stack local variables to see them. To learn about the meaning of these registers, you need to consult the user's guide for your CPU and Appendix C, "*Architectures*" on page 337.

For your convenience, TotalView displays the bit settings of many CPU registers symbolically. For example, TotalView symbolically displays registers that control rounding and exception enable modes. You can edit the values of these registers and continue execution of your program. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are displayed vary from platform to platform, see Appendix C, "*Architectures*" on page 337 for information on the registers supported for your CPU. For general information on editing the value of variables (including registers), refer to "*Displaying Areas of Memory*" on page 156.





## Chapter 7

# Examining and Changing Data

This chapter explains how to examine and change data as you debug your program. You will learn how to:

- Displaying Variable Windows
- Diving in Variable Windows
- Changing the Values of Variables
- Changing the Data Type of Variables
- Working with Opaque Data
- Changing the Address of Variables
- Changing Types to Display Machine Instructions
- Displaying C++ Types
- Displaying Fortran Types
- Displaying Thread Objects

## Displaying Variable Windows

You can create windows that display local variables, registers, global variables, areas of memory, and machine instructions.

### Displaying Local Variables and Registers

In the Stack Frame Pane, you can dive into a formal parameter, local variable, or register to display a Variable Window. You can also dive into formal parameters and local variables in the Source Pane. The Variable Window lists the name, address, data type, and value for the object, as shown in Figure 77.

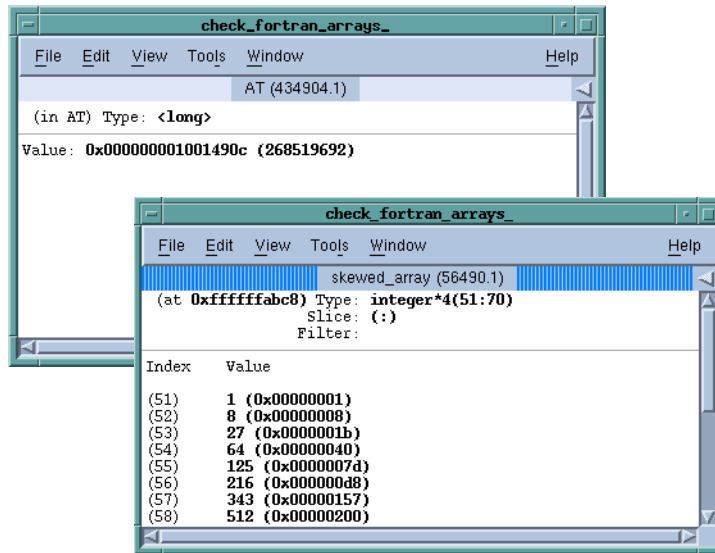


FIGURE 77: Diving into Local Variables and Registers

The top window is for a register while the bottom window is for a local array variable.

You can also display a local variable by using the **View > Lookup Variable** command. When prompted, enter the name of the variable in the dialog box.

If Variable Windows remain open while a process or thread runs, TotalView updates the information in the windows when the process or thread stops. If TotalView cannot find a stack frame for a displayed local variable, it displays **Stale** in the pane's header to warn you that you cannot trust the data, since the variable no longer exists.

When you debug recursive code, TotalView does not automatically refocus a Variable Window onto different instances of a recursive function. If you have a breakpoint in a recursive function, you may need to explicitly open a new Variable Window to see the value of a local variable for that stack frame.

## Displaying a Global Variable

You can display a global variable by:

- Diving into the variable in the Source Pane.
- Select the **View > Lookup Variable** command. When prompted, enter the name of the variable.

A Variable Window appears for the global variable, as shown in the following figure.

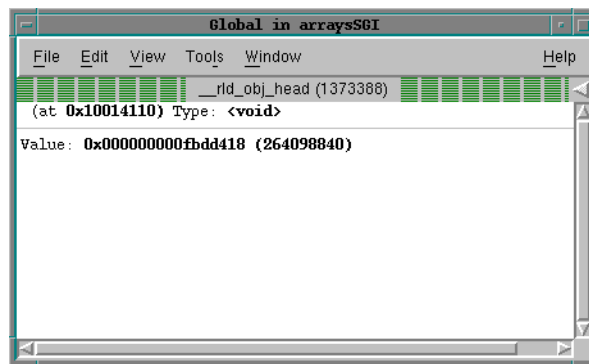


FIGURE 78: Variable Window for a Global Variable

## Displaying All Global Variables

The Process Window's **Tools > Globals** command displays all of the current process's global variables. This window contains the name and value of every global variable used by the process, as shown in Figure 79.

## Displaying Long Variable Names

If due to space limitations, TotalView cannot display all the characters in a variable name, it inserts ellipses (...) to indicate the name was truncated. Typically, this occurs when it is displaying C++ names that have been demangled or STL variables. Figure 80 on page 157 shows a Variable Window containing a series of STL names. The two additional windows are what TotalView displays when you click on the ellipses. Notice that one of the windows has an **Apply** button. This indicates that the field is editable.

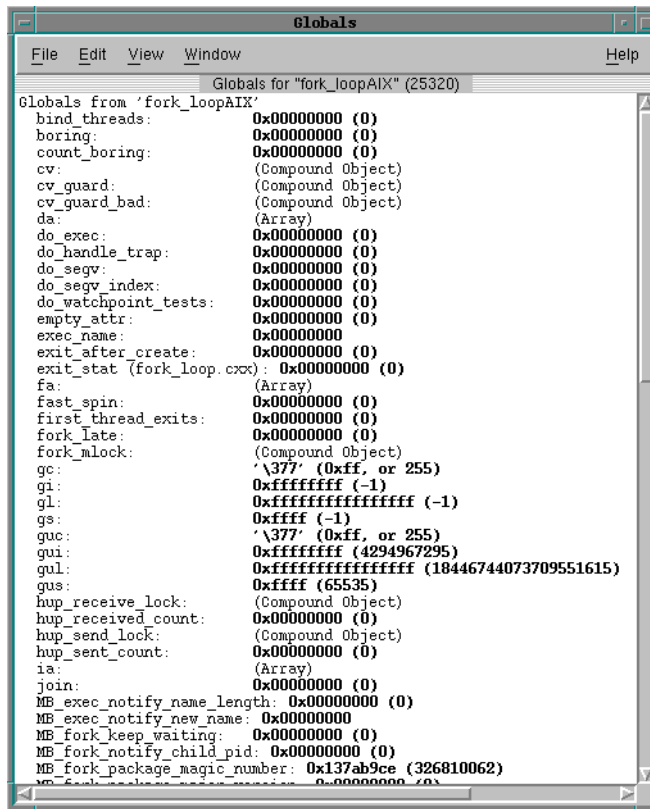


FIGURE 79: Global Variables Window

## Displaying Areas of Memory

You can display areas of memory in hexadecimal and decimal. Do this by selecting the **View > Lookup Variable** command and then enter one of the following in the dialog box:

### ■ A hexadecimal address

When you enter a single address, TotalView displays the word of data stored at that address.

### ■ A pair of hexadecimal addresses

When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

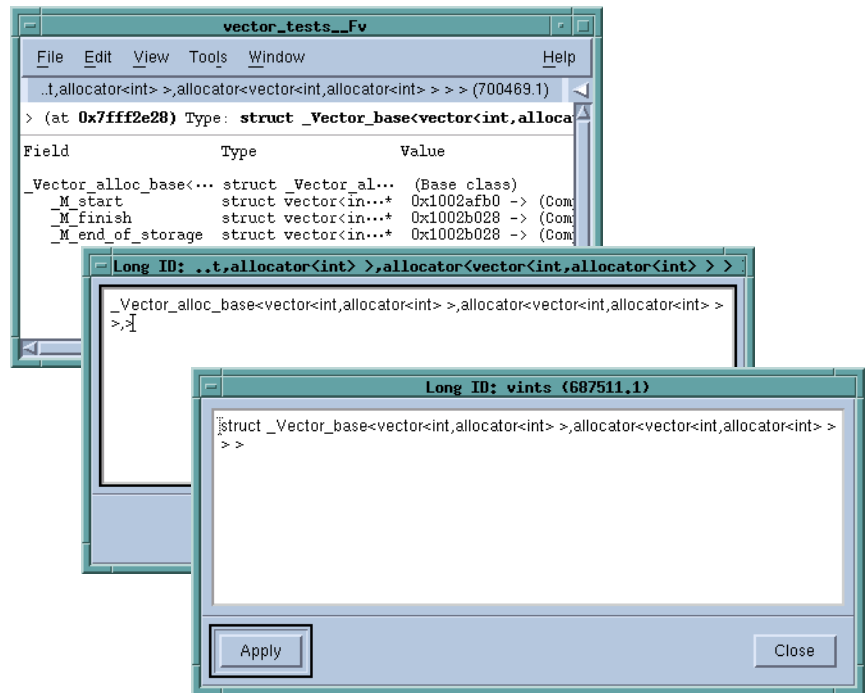


FIGURE 80: Displaying Long STL Names

**NOTE** All hexadecimal constants must have a "0x" prefix. Also, you can enter these addresses using expressions.

The Variable Window for an area of memory, as shown in Figure 81, displays the address and contents of each word.

The starting location of the memory area is displayed at the top of the window's data area. Within the window, information is displayed in hexadecimal and in decimal.

## Displaying Machine Instructions

You can display the machine instructions for entire routines as follows:

- Dive into the address of an assembler instruction in the Source Pane (such as **main+0x10** or **0x60**). A Variable Window displays the instructions for the entire function and highlights the instruction into which you dived.

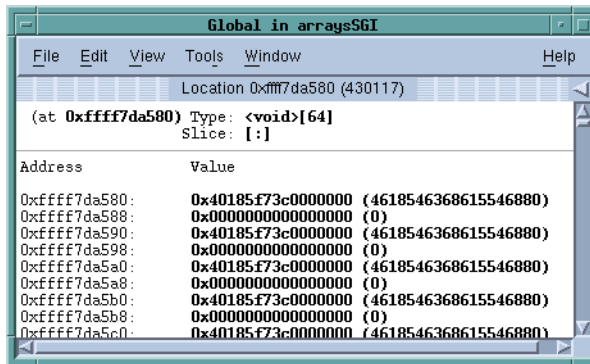


FIGURE 81: Variable Window for Area of Memory

- Dive into the PC in the Stack Frame Pane. A Variable Window lists the instructions for the entire function containing the PC, and highlights the instruction to which the PC points.

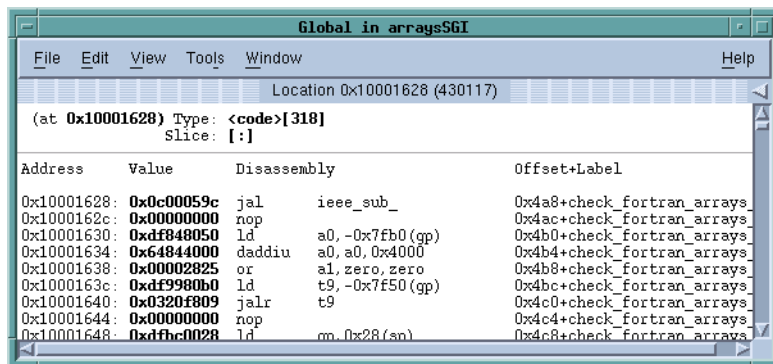


FIGURE 82: Variable Window with Machine Instructions

- Cast a variable to type `<code>` or array of `<code>`, as described in "Changing Types to Display Machine Instructions" on page 171. (See Figure 83.)

## Closing Variable Windows

When you are finished analyzing the information in a Variable Window, use the **File > Close** command to close the window. You can also use the **File > Close Similar** command to close all Variable Windows.

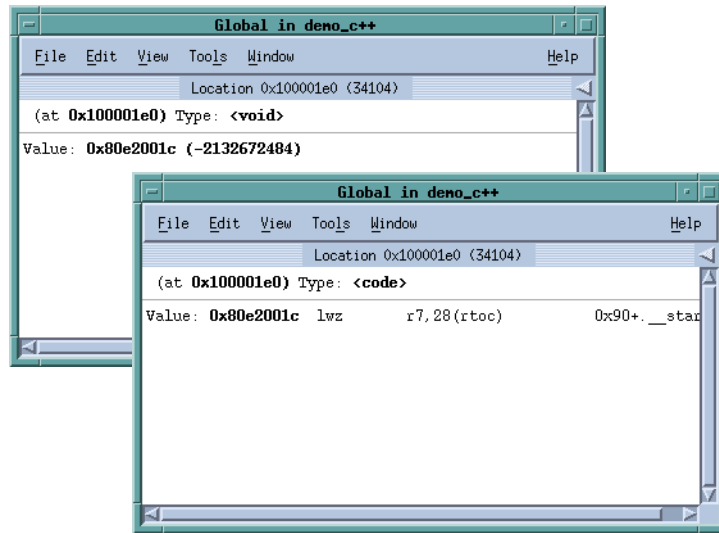


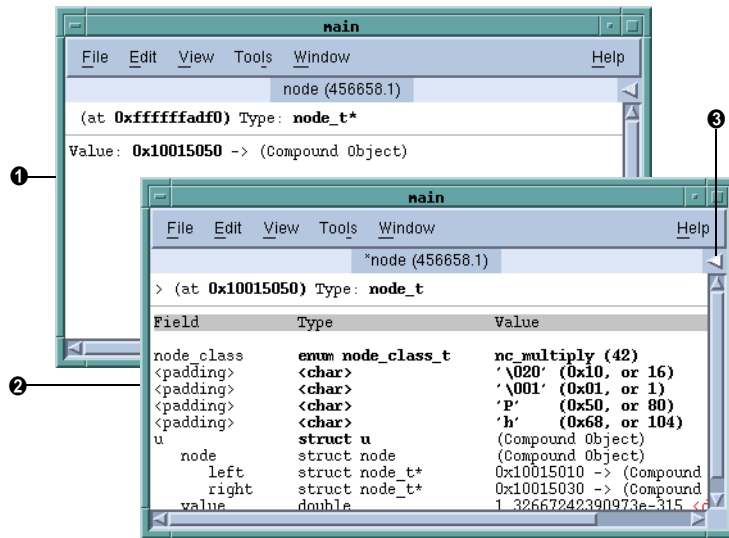
FIGURE 83: Casting Code

## Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive into the contents listed in the Variable Window. This additional dive is called a *nested dive*. When you perform a nested dive, the Variable Window replaces the original information with information about the current variable. With nested dives, the original Variable Window is known as the *base window*.

You can select the left pointing triangular icon in the top right corner of this window to restore the window to what it was before the last dive operation. You can reselect this icon as often as you need so to restore the Variable Window to its original display.

Figure 84 shows the results of diving into a variable in the Stack Frame Pane of `main()`. This example dives into a pointer variable named `node` with a type of `node_t*`. The first Variable Window (the base window) displays the value of `node`.



- ❶ Base window: First dive (on the variable `node_t*`, a pointer)
- ❷ Nested window: Second dive (on the value of `node_t*`)
- ❸ Undive icon

FIGURE 84: Nested Dives

Diving into this value tells TotalView to replace the window with a *nested dive window*. That is, new data replaces the old data without TotalView creating a new window. The nested dive window—displayed in the bottom right corner of the figure—shows the structure referenced by the `node_t*` pointer.

TotalView maintains each dive on a dive stack.

You can manipulate Variable Windows and nested dive windows in the following ways:

- To “undive” from a nested dive, select the left-facing arrow in the upper right-hand corner of the Variable Window. After clicking on the arrow, the previous contents of the Variable Window appears.
- If you have performed several nested dives and want to create a new copy of the base window, select the **Window > Duplicate Base** command.
- If you dive into a variable that already has a Variable Window open, the Variable Window pops to the surface.



- If you select the **Window > Duplicate** command, a new Variable Window appears that is a duplicate of the current Variable Window except that it has an empty dive stack.

## Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window by selecting the value and typing the new value. In addition to typing a value, you can also type an expression. For example, you can enter  $1024 * 1024$  as shown in Figure 85. This expression can use logical operators.

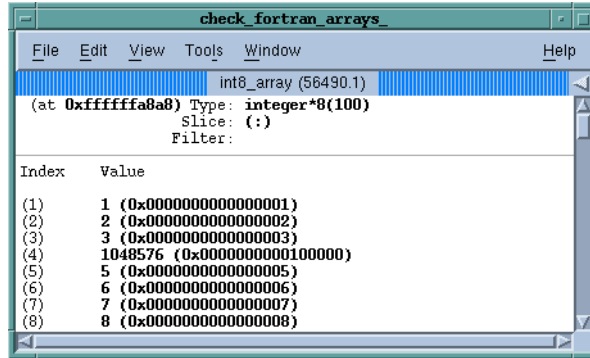


FIGURE 85: Using an Expression to Change a Value

You can also edit a variable's value directly within the Stack Frame Pane by selecting it. You cannot, however, change the value of bit fields directly, but you can use the **Tools > Evaluation** Window to assign a value to a bit field. See "Evaluating Expressions" on page 232. Similarly, you cannot directly change the value of fields in nested structures: you must first dive into the value.

## Changing the Data Type of Variables

The data type declared for the variable determines its format and size (amount of memory). For example, if you declare an **int** variable, TotalView displays the variable as an integer.

You can change the way TotalView displays data in the Variable Window by editing its data type. This is known as *casting*. TotalView assigns types to all data types, and in most cases, they are identical to their programming language counterparts.

- When displaying a C variable, TotalView data types are identical to C type representations, except for pointers to arrays. TotalView uses a simpler syntax for pointers to arrays.
- When displaying a Fortran variable, TotalView types are identical to Fortran type representations for most data types including **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, and **CHARACTER**.

If the window contains a structure with a list of fields, you can edit the types of the fields listed in the window.

**NOTE** When you edit a type, TotalView changes how it displays the variable in the current Variable Window, but other windows listing the variable remain the same.

## How TotalView Displays C Data Types

TotalView's syntax for displaying data is identical to C Language cast syntax for all data types except pointers to arrays. That is, you should use C Language cast syntax for **int**, **short**, **unsigned**, **float**, **double**, **union**, and all named **struct** types.

TotalView types are read from right to left. For example, **<string>\*[20]\*** is a pointer to an array of 20 pointers to **<string>**.

Table 11 shows some common types.

TABLE 11: Common Types

Type String	Meaning
<b>int</b>	Integer
<b>int*</b>	Pointer to integer
<b>int[10]</b>	Array of 10 integers
<b>&lt;string&gt;</b>	Null-terminated character string
<b>&lt;string&gt; **</b>	Pointer to a pointer to a null-terminated character string
<b>&lt;string&gt; *[20]*</b>	Pointer to an array of 20 pointers to null-terminated strings

You can also enter C Language cast syntax verbatim in the type field for any type. In addition, TotalView can display C Language cast syntax permanently if you set an X Window Resource. See **totalview\*cTypeStrings** on page 277 for further information.

The following sections discuss the more complex types.

## Pointers to Arrays

Suppose you declared a variable **vbl** as a pointer to an array of 23 pointers to an array of 12 objects of type **mytype\_t**. The C language declaration for this is:

```
mytype_t (*vbl)[23][12];
```

Here is how you would cast the **vbl** variable to this type:

```
(mytype_t (*)(*)[23][12])vbl
```

The TotalView cast for **vbl** is:

```
mytype_t[12]*[23]*
```

## Arrays

Array type names can include a lower and upper bound separated by a colon (:).

By default, the lower bound for a C or C++ array is 0, and the lower bound for Fortran is 1. In the following example, an array of ten integers is declared in C and then in Fortran:

```
int a[10];
integer a(10)
```

The elements of the array range from **a[0]** to **a[9]** in C, while the elements of the equivalent Fortran array range from **a(1)** to **a(10)**.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements) of the dimension. Consider the following array declaration in Fortran:

```
integer a(1:7,1:8)
```

Since both dimensions of the array use the default lower bound for Fortran (1), TotalView displays the data type of the array by using only the extent of each dimension, as follows:

```
integer(7,8)
```

If an array declaration does not use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, you would declare an array of integers with the first dimension ranging from -1 to 5 and the second dimension ranging from 2 to 10, as follows:

```
integer a(-1:5,2:10)
```

TotalView displays this in exactly the same way.

When editing a dimension of an array, you can enter just the extent (if using the default lower bound) or both the lower and upper bounds separated by a colon.

TotalView also lets you display a subsection of an array, or filter a scalar array for values matching a filter expression. Refer to “*Displaying Array Slices*” on page 183 and “*Array Data Filtering*” on page 188 for further information.

## Typedefs

TotalView recognizes the names defined with **typedef**, but displays the definition of the type (that is, the base data type), rather than its name. For example:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

The debugger displays the type for **p\_vbl** as **double\***, not as **dptr\_t**.

## Structures

TotalView treats **struct** as a keyword. You can type **struct** as part of the type string, but it is optional. If you have a structure and another data type with the same name, however, you must include **struct** with the name of the structure so TotalView can distinguish between the two data types.

If you name a structure using **typedef**, the debugger uses the **typedef** name as the type string. Otherwise, the debugger uses the structure tag for the **struct**.

For example, consider the structure definition:

```
typedef struct mystruc_struct {
    int field_1;
    int field_2;
} mystruc_type;
```

TotalView displays **mystruc\_type** as the type for **struct mystruc\_struct**.

TotalView does not interpret the definition of structures in a type string. For example, it cannot interpret a definition such as **struct {int a; int b;}.**

## Unions

TotalView displays a union in the same way that it displays a structure. Even though the fields of a union are overlaid in storage, TotalView displays them on separate lines in the Variable Window.

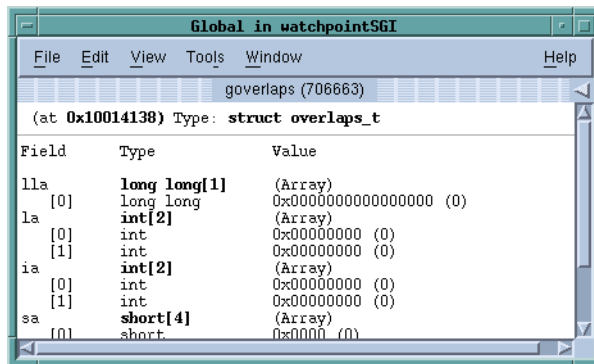


FIGURE 86: Displaying a Union

When TotalView displays some complex arrays and structures, it displays the compound object or array types in the Variable Window.

**NOTE** Editing the compound object or array types could yield undesirable results. We do not recommend editing these types.

## Built-In Types

TotalView provides a number of predefined types. These types are enclosed in angle brackets (<>) to avoid conflict with types defined in a programming language. You can use these built-in types anywhere a user-defined type can be used, such as in a cast expression. These types are also useful when debugging executables with no debugging symbol table information. The following table lists the built-in types.

TABLE 12: Built-In Types

Type String	Language	Size	Meaning
<address>	C	void*	Void pointer (address)
<char>	C	char	Character
<character>	Fortran	character	Character
<code>	C	parcel	Machine instructions A parcel is the number of bytes required to hold the shortest instruction for the target architecture.
<complex>	Fortran	complex	Single-precision floating-point complex number. <b>complex</b> types contain a real part and an imaginary part, which are both of type <b>real</b> .
<complex*8>	Fortran	complex*8	<b>real*4</b> -precision floating-point complex number <b>complex*8</b> types contain a real part and an imaginary part, which are both of type <b>real*4</b> .
<complex*16>	Fortran	complex*16	<b>real*8</b> -precision floating-point complex number <b>complex*16</b> types contain a real part and an imaginary part, which are both of type <b>real*8</b> .
<double>	C	double	Double-precision floating-point number

TABLE 12: Built-In Types (cont.)

Type String	Language	Size	Meaning
<double precision>	Fortran	double precision	Double-precision floating-point number
<extended>	C	long double	Extended-precision floating- point number  Extended-precision numbers must be supported by the tar- get architecture.
<float>	C	float	Single-precision floating-point number
<int>	C	int	Integer
<integer>	Fortran	integer	Integer
<integer*1>	Fortran	integer*1	One-byte integer
<integer*2>	Fortran	integer*2	Two-byte integer
<integer*4>	Fortran	integer*4	Four-byte integer
<integer*8>	Fortran	integer*8	Eight-byte integer
<logical>	Fortran	logical	Logical
<logical*1>	Fortran	logical*1	One-byte logical
<logical*2>	Fortran	logical*2	Two-byte logical
<logical*4>	Fortran	logical*4	Four-byte logical
<logical*8>	Fortran	logical*8	Eight-byte logical
<long>	C	long	Long integer
<long long>	C	long long	Long long integer
<real>	Fortran	real	Single-precision floating-point number
<real*4>	Fortran	real*4	Four-byte floating-point num- ber
<real*8>	Fortran	real*8	Eight-byte floating-point num- ber
<real*16>	Fortran	real*16	Sixteen-byte floating-point number
<short>	C	short	Short integer

TABLE 12: Built-In Types (cont.)

Type String	Language	Size	Meaning
<string>	C	char	Array of characters
<void>	C	long	Area of memory

The following sections contain more information the following built-in types:

- Character arrays (<string> Data Type)
- Areas of memory (<void> Data Type)
- Instructions (<code> Data Type)

### Character arrays (<string> Data Type)

If you declare a character array as **char vbl[n]**, TotalView automatically changes the type to **<string>[n]**; that is, a null-terminated, quoted string with a maximum length of *n*. Thus, by default, the array is displayed as a quoted string of *n* characters, terminated by a null character. Similarly, TotalView changes **char\*** declarations to **<string>\*** (a pointer to a null-terminated string).

Since most C character arrays represent strings, the **<string>** type can be very convenient. If, however, you intended the **char** data type to be a pointer to a single character or an array of characters, you can edit the **<string>** back to a **char** (or **char[n]**) to display the variable as you declared it.

### Areas of memory (<void> Data Type)

TotalView uses the **<void>** type for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The **<void>** type is similar to the **int** in the C language.

If you dive into registers or display an area of memory, TotalView lists the contents as a **<void>** data type. Further, if you display an array of **<void>** variables, the index for each object in the array is the address, not an integer. This address can be useful when displaying large areas of memory.



If desired, you can change a **<void>** into another type. Similarly, you can change any type into a **<void>** to see the variable in decimal and hexadecimal.

### Instructions (**<code>** Data Type)

TotalView uses the **<code>** data type to display the contents of a location as machine instructions. Thus, to look at disassembled code stored at a location, dive on the location and change the type to **<code>**. To specify a block of locations, use **<code>[*n*]**, where *n* is the number of locations being displayed.

## Type Casting Examples

This section contains some common type casting examples, as follows:

- Displaying the argv Array
- Displaying Declared Arrays
- Displaying Allocated Arrays

### Displaying the argv Array

Typically, **argv** is the second argument passed to **main()**, and it is either a **char \*\*argv** or **char \*argv[ ]**. Since these declarations are equivalent (a pointer to one or more pointers to characters), TotalView converts both to **<string>\*\*** (a pointer to one or more pointers to null-terminated strings).

Suppose **argv** points to an array of three pointers to character strings. Here is how you can edit its type to display an array of three pointers:

- 1 Select the type string for **argv**.
- 2 Edit the type string using the field editor commands. Change it to:  
**<string>\*[3]\***
- 3 To display the array, dive into the value field for **argv**. (See Figure 87.)

### Displaying Declared Arrays

TotalView displays arrays in the same way as it displays local and global variables. In the Stack Frame or Source Pane, dive into the declared array. A Variable Window displays the elements of the array.

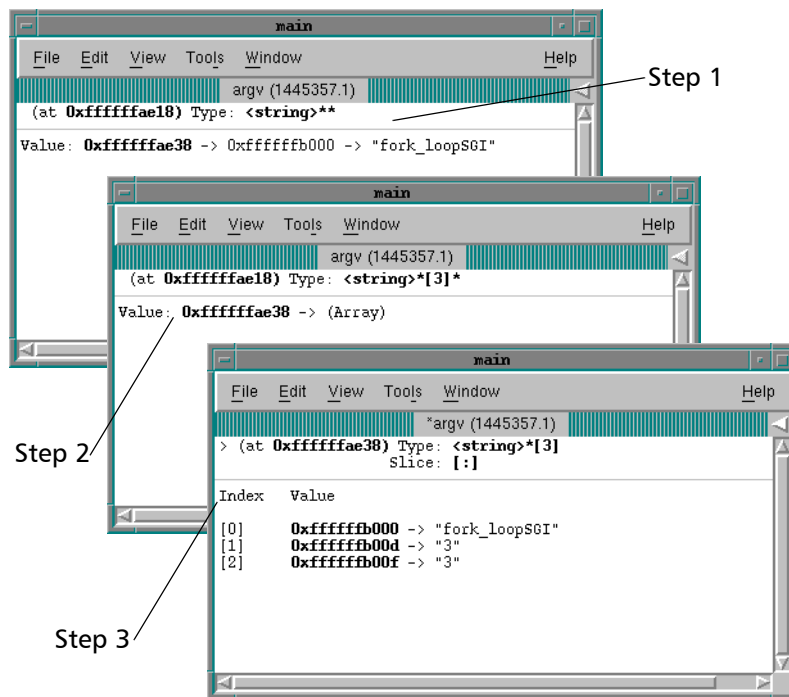


FIGURE 87: Editing argv

## Displaying Allocated Arrays

The C language uses pointers for dynamically allocated arrays. For example:

```
int *p = malloc(sizeof(int) * 20);
```

Because TotalView does not know that `p` actually points to an array of integers, here is how you would display the array:

- 1 Dive on the variable `p` of type `int*`.
- 2 Change its type to `int[20]*`.
- 3 Dive on the value of the pointer to display the array of 20 integers.

## Working with Opaque Data

An opaque type is a data type that is not fully specified, is hidden, or whose declaration is deferred. For example the following C declaration defines the data type for **p** as pointer to **struct foo**, which is not yet defined:

```
struct foo;
struct foo *p;
```

When TotalView encounters this kind of information, it indicates its data type by appending **<opaque>** to the declaration. For example:

```
struct foo <opaque>
```

If the type is actually defined in another module, deleting **<opaque>** from the data type tells TotalView to find the actual definition for the type.

On platforms where TotalView uses “lazy reading” of the symbol table, you must force TotalView to read the symbols from the module containing the full type definition of the opaque type. Use the **View > Lookup Function** command to force TotalView to read the symbols, as described in “*Finding the Source Code for Functions*” on page 127.

## Changing the Address of Variables

You can edit the address of a variable in a Variable Window. When you edit the address, the Variable Window shows the contents of the new location.

You can also enter an address expression, such as **0x10b8 – 0x80**.

## Changing Types to Display Machine Instructions

Here is how you can display machine instructions in any Variable Window:

- 1 Select the type string at the top of the Variable Window.
- 2 Change the type string to be an array of **<code>** data types, where *n* indicates the number of instructions to be displayed. For example:

```
<code>[n]
```

TotalView displays the contents of the current variable, register, or area of memory as machine-level instructions.

The Variable Window (shown in Figure 82 on page 158) lists the following information about each machine instruction:

<b>Address</b>	The machine address of the instruction.
<b>Value</b>	The hexadecimal value stored in the location.
<b>Disassembly</b>	The instruction and operands stored in the location.
<b>Offset + Label</b>	The symbolic address of the location as a hexadecimal offset from a routine name.

You can also edit the value listed in the **Value** field for each machine instruction.

## Displaying C++ Types

### Classes

TotalView displays C++ classes and accepts **class** as a keyword. When you debug C++, TotalView also accepts the *unadorned* name of a **class**, **struct**, **union**, or **enum** in the type field. TotalView displays nested classes that use inheritance, showing derivation by indentation.

**NOTE** Some C++ compilers do not output accessibility information. In these cases, the information is omitted from the display.

For example, the following figure displays an object of a **class c**:

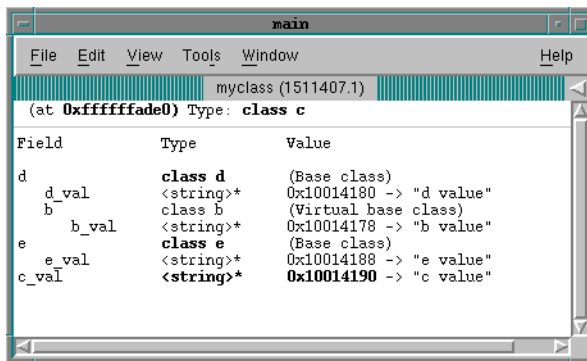


FIGURE 88: Displaying C++ Classes That Use Inheritance

The definition is as follows:

```
class b {
    char * b_val;
public:
    b() {b_val = "b value";}
};

class d : virtual public b {
    char * d_val;
public:
    d() {d_val = "d value";}
};

class e {
    char * e_val;
public:
    e() {e_val = "e value";}
};

class c : public d, public e {
    char * c_val;
public:
    c() {c_val = "c value";}
};
```

## Changing Class Types in C++

TotalView tries to display the correct data when you change the type of a Variable Window to move up or down the derivation hierarchy.

If a change in the data type also requires a change in the address of the data being displayed, TotalView asks you about changing the address. For example, if you edit the type field in **class c** shown in Figure 89 to **class e**, TotalView displays the following dialog box:

Selecting **Yes** tells TotalView to change the address to ensure that it displays the correct base class member. Selecting **No** tells TotalView to display the memory area as if it were an instance of the type to which it is being cast, leaving the address unchanged.

Similarly, if you change a data type in the Variable Window so you can cast a base class to a derived class, and that change requires an address change,

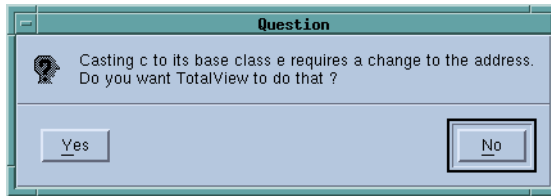


FIGURE 89: C++ Type Cast to Base Class Dialog Box

the debugger asks you to confirm the operation. For example, Figure 90 shows the dialog posted if you cast from `class e` to `class c`:

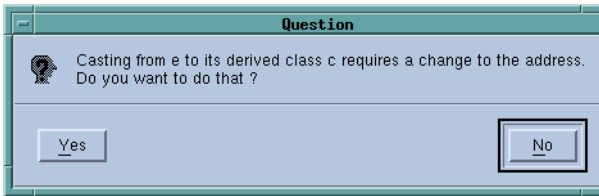


FIGURE 90: C++ Type Cast to Derived Class Dialog Box

## Displaying Fortran Types

TotalView allows you to display FORTRAN 77 and Fortran 90 data types.

### Displaying Fortran Common Blocks

For each common block defined within the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The Stack Frame Pane displays the name of each common block for a function. The names of common block members have function scope, not global scope.

TotalView creates a user-defined data type for the common block in which each of the common block members are fields in the type. If you dive on a common block name in the Stack Frame Pane, TotalView displays the entire common block in a Variable Window, as shown in Figure 91.

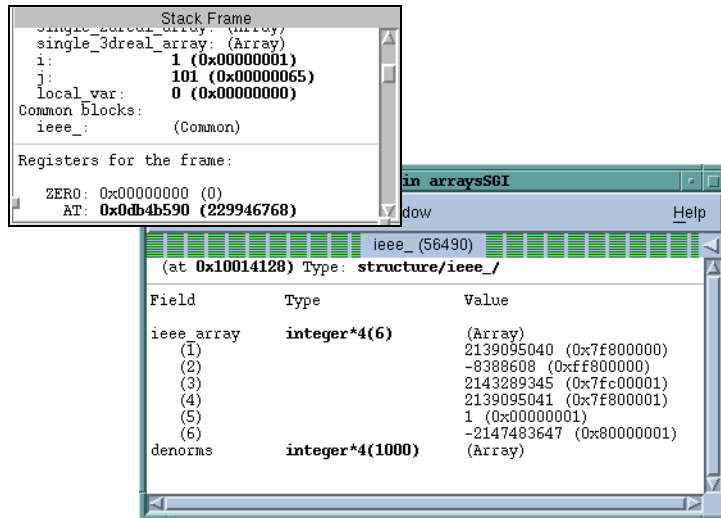


FIGURE 91: Diving into a Common Block List in the Stack Frame Pane

The top-left pane shows a common block list in a Stack Frame Pane. The bottom right window shows the results of diving on the common block to see its elements.

If you dive on a common block member name, TotalView searches all common blocks for a matching member name and displays the member in a Variable Window.

Normally, TotalView displays the initial address for a common block in the Variable Window. When the common block is a composite object with separate addresses for each component, TotalView displays a **Multiple** tag to indicate that it cannot display a single address.

## Displaying Fortran Module Data

TotalView tries to locate all data associated with a Fortran module and provide a single display that contains all of it. For functions and subroutines defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

**NOTE** TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView may restrict what is displayed.

Although a function may use a module, TotalView may not be able to determine if the module was used or what the true names of the variables in the module are. In this case, either module variables appear as local variables of the subroutine, or a module appears on the list of modules in the Stack Frame Pane that contains (with renaming) only the variables used by the subroutine.

Alternatively, you can view a list of all the known modules by using the **Tools > Fortran Modules** command. Like in any Variable Window, you can dive through an entry to display the actual module data, as shown in Figure 92.



FIGURE 92: Fortran Modules Window



**NOTE** If you are using the SUNPro compiler, TotalView can only display module data if you force it to read the debug information for a file that contains the module definition or a module function. For more information, see “*Finding the Source Code for Functions*” on page 127.

## Debugging Fortran 90 Modules

Fortran 90 and Fortran 95 let you place functions, subroutines, and variables inside modules. These modules can then be **USED** (included) elsewhere. When modules are **USED**, the names in the module become available in the *using* compilation unit, unless they have been excluded by **USE ONLY**, or renamed. This means that you do not need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you will need to know the location of the function being called. Consequently, TotalView uses the following syntax when it displays a function contained within a module:

*modulename`functionname*

You can use also this syntax in the **File > New Program** and **View > Lookup Variable** commands.

Fortran 90 also introduced the idea of a contained function that is only visible in the scope of its parent and siblings. There can be many contained functions in a program, all using the same name. If the compiler gave TotalView the function name for a nested functions, TotalView displays it using the following syntax:

*parentfunction() `containedfunction*

Within contained functions, all of the parent function’s variables are visible and accessible through a static chain. If the compiler retained information about the static chain, TotalView can access these variables in the same way as the compiled code does. Consequently, they are visible in Variable Windows, and from evaluation points or expressions. If the compiler does not pass on information about the static chain, TotalView can still find these up-level variables and display them in Variable Windows, but you will not be able to use them in evaluation points or expressions.

## Fortran 90 User-Defined Type

A Fortran 90 user-defined type is similar to a C structure. TotalView displays a user-defined type as **type(name)**, which is the same syntax used in Fortran 90 to create a user-defined type. For example, here is a code fragment that defines a variable **matrix1** of **type(sparse)**:

```
TYPE WHOPPER
  LOGICAL, DIMENSION(ISIZE) :: FLAGS
  DOUBLE PRECISION, DIMENSION(ISIZE) :: DPSA
  DOUBLE PRECISION, DIMENSION(:), POINTER :: DPPA
END TYPE WHOPPER
```

```
TYPE(WHOPPER), DIMENSION(:), ALLOCATABLE :: TYP2
```

TotalView displays this code as shown in Figure 93.

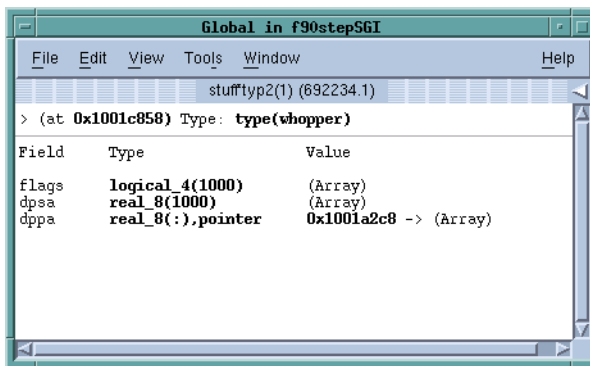


FIGURE 93: Fortran 90 User Defined Type

## Fortran 90 Deferred Shape Array Type

Fortran 90 allows you to define deferred shape arrays and pointers. The actual bounds of the array are not determined until the array is allocated, the pointer is assigned, or (in the case of an assumed shape argument to a subroutine) the subroutine is called. The type of deferred shape arrays is displayed by TotalView as **type(:)**, which is the same way that you declared the array.

When TotalView displays the data for a deferred shape array, it displays the type used in the definition of the variable and the actual type that this in-

stance of the variable has. The actual type is not editable since you can achieve the same effect by editing the definition's type. The following example shows the type of a deferred shape rank 2 array of **real** data with runtime lower bounds of -1 and 2, and upper bounds of 5 and 10:

```
Type: real(:, :)
Actual Type: real(-1:5, 2:10)
Slice: (:, :)
```

## Fortran 90 Pointer Type

A Fortran 90 pointer type allows you to point to scalar or array types. The debugger displays pointer types as *type,pointer*, which is the same syntax used in Fortran 90 to create a pointer variable.

For example, a **pointer** to a rank 1 deferred shape array of **real** data is displayed in the Variable Window as:

```
Type: real(:), pointer
```

To view the data instead of the pointer variable, dive on the value.

**NOTE** If you are using the IBM xlf compiler, TotalView cannot determine the rank of the array from the debugging information. In this case, the type of a pointer to an array appears as "type(...),pointer". The actual rank is filled in when you dive through the pointer to look at the data.

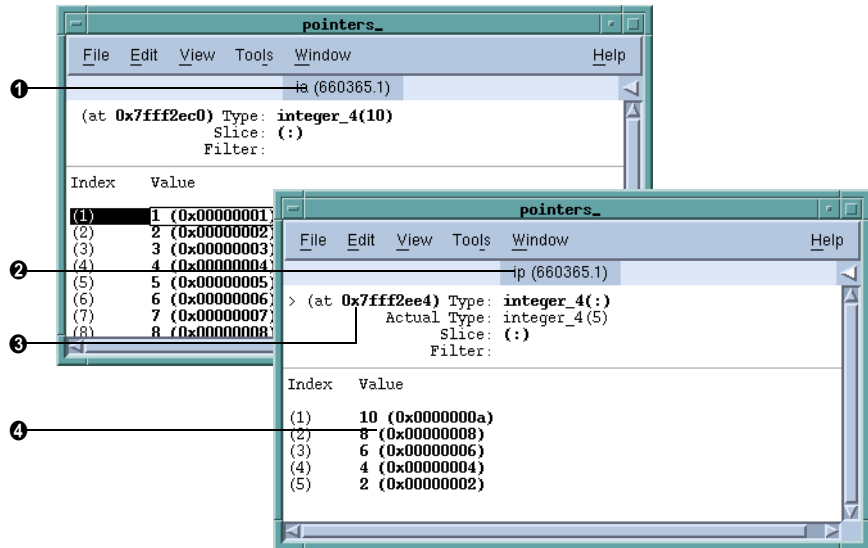
The value of the pointer is displayed as the address of the data to which the pointer points. This address is not necessarily the array element with the lowest address.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that indices and values it displays in a Variable Window are the same as you would see in the Fortran code.

For example:

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = 1, 10
    ia(i) = i
end do
ip => ia(10:1:-2)
```

After diving through the **ip** pointer, TotalView displays the window shown in Figure 94.



- ❶ Target array ia
- ❷ Pointer ip into array ia
- ❸ Address of ip(1)
- ❹ Values reflect slice

FIGURE 94: Fortran 90 Pointer Value

Notice that the address displayed is not that of the array's base. Since the array's stride is negative, succeeding array elements are at lower absolute addresses. Consequently, the address displayed is that of the array element having the lowest index (which may not be the first displayed element if you used a slice to display the array with reversed indices).

## Displaying Fortran PARAMETERS

A Fortran **PARAMETER** defines a named constant. Most compilers do not generate information that TotalView can use. With a few changes to your program, you can see this kind of information.

If you are using Fortran 90, you can define variables in a module that you initialize to the value of these **PARAMETER** constants. For example:

```
INCLUDE 'PARAMS.INC'
```

```
MODULE CONSTS
```

```
  SAVE
```

```
  INTEGER PI_C = PI
```

```
  ...
```

```
END MODULE CONSTS
```

If you compile and link this module into your program, the value of its variables are visible.

If you are using Fortran 77, you could achieve the same results if you make the assignments in a common block and then include the block in **main()**. You would also use a block data subroutine to access this information.

## Displaying Thread Objects

On Compaq Tru64 UNIX and IBM AIX systems, TotalView can display information about mutexes and conditional variables. In addition, TotalView can display information on read/writes locks and data keys on IBM AIX. You can obtain this information by selecting the **Tools > Thread Objects** command. After selecting this command, TotalView displays a window that will either contain two tabs (Compaq) or four tabs (IBM). Figure 95 on page 182 shows some AIX examples.

Diving on an any line within these windows displays a Variable containing additional information about the item.

Here are some things you should know:

- If you are displaying data keys, many applications initially set keys to zero (the NULL pointer value). TotalView does not display a key's information, however, until a thread sets a non-NULL value to the key.
- If you select a thread ID within a data key window, you can dive on it using the **View > Dive Thread** and **View > Dive Thread New** commands to display a Process Window for that thread ID.

The Help contains considerable information on the contents of these windows.

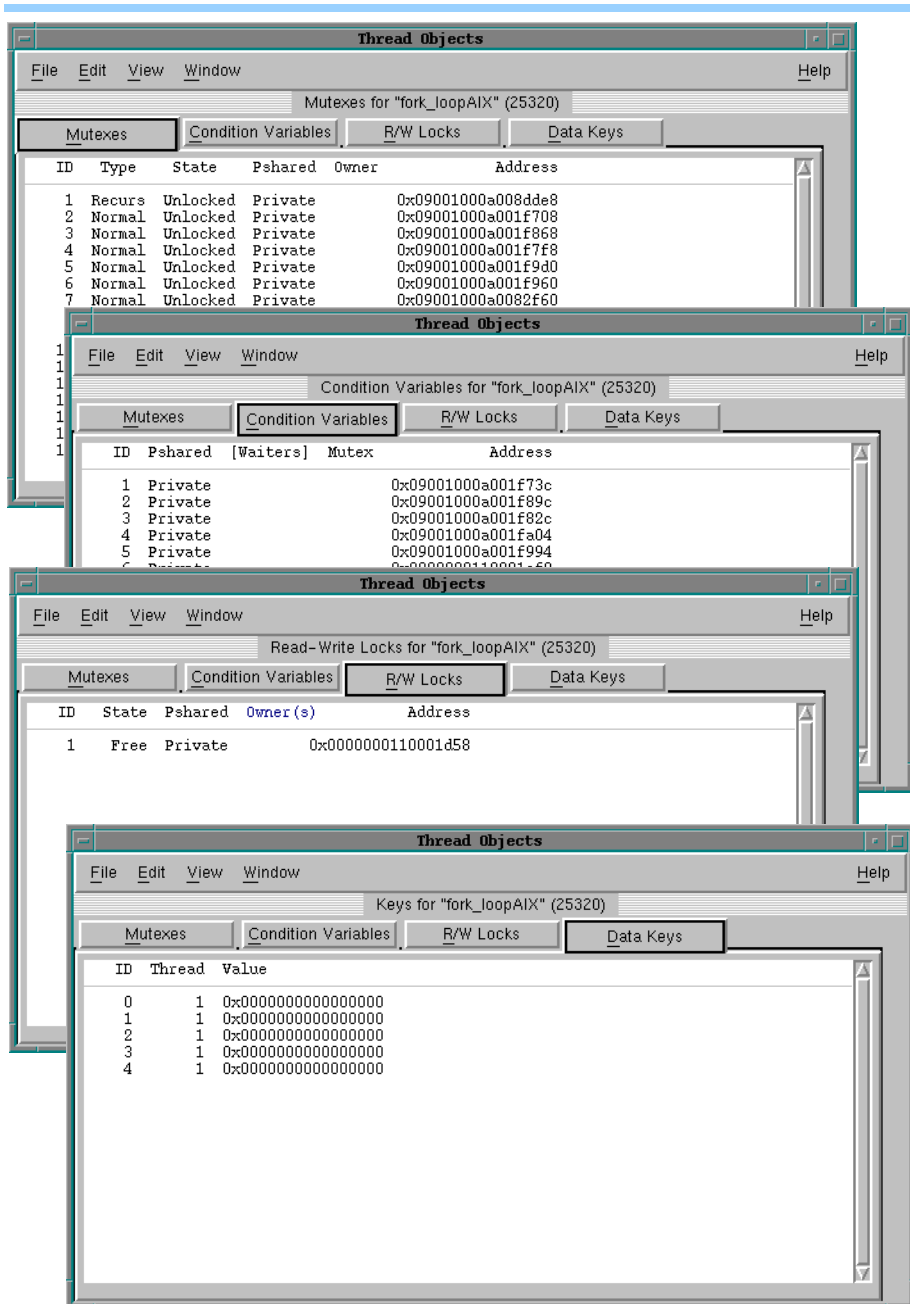


FIGURE 95: Thread Objects Pages

# Examining Arrays

This chapter explains how to examine and change data as you debug your program. You will learn about the following:

- Examining and Analyzing Arrays
- Displaying a Variable in All Processes or Threads
- Visualizing Array Data

## Examining and Analyzing Arrays

TotalView can quickly display very large arrays in Variable Windows. If an array overlaps nonexistent memory, the initial portion of the array is correctly formatted. If memory is not allocated for an array element, TotalView displays **Bad Address** in the element's subscript.

## Displaying Array Slices

TotalView lets you display array subsections by editing the *slice field* within an array's Variable Window. (An array subsection is called a *slice*.) The slice field contains placeholders for all array dimensions. For example, here is a C declaration for a three-dimensional array:

```
integer ia[10][20][5]
```

TotalView defines this slice as `[:][:][:]`.

Here is a Fortran 90 deferred shape array definition:

```
integer, dimension (:,:) :: ia
```

Its TotalView slice definition is `(:,:)`.

As you can see, TotalView displays as many colons (:) as there are array dimensions. Initially, the field contains [:] for C arrays or (:) for Fortran arrays.

## Slice Definitions

A slice definition has the following form:

*lower\_bound:upper\_bound:stride*

This tells TotalView to display every *stride* element of the array, starting at the *lower\_bound* and continuing through the *upper\_bound*, inclusive. (A *stride* tells TotalView that it should skip over elements and not display them.)

For example, if you specified a slice of [0:9:9] for a 10-element C array, TotalView displays the first element and last element, which is the ninth element beyond the lower bound.

If a slice is defined as [lb:ub:stride], TotalView represents the set of values of *i* generated by the **append** statements in the following way:

```
i = lb
if (stride > 0)
    while ( i <= ub )
        append i
        i = i + stride
else
    while ( i >= ub )
        append i
        i = i + stride
```

If **stride** < 0 and **ub** > **lb**, TotalView treats the slice as if it were:

[ub : lb : stride]

(This is an extension to the way Fortran displays slices.) Consequently, TotalView lets you view a dimension with reversed indexing. For example, the following definition tells TotalView to display an array beginning at its last value and moving to its first:

[::-1]

In contrast, Fortran 90 requires that you explicitly enter the upper and lower bounds when you are reversing the order in which it displays array elements.



Because the default value for the stride is 1, you can omit the stride (and the colon that precedes it) if your stride value is 1. For example, the following two definitions display array elements 0 through 9:

```
[0:9:1]
[0:9]
```

If the lower and upper bound are the same number, you can specify the slice with just a single number. This number indicates the lower and upper bound. For example, the following two definitions tell TotalView to display array element 9:

```
[9:9:1]
[9]
```

**NOTE** The `lower_bound`, `upper_bound`, and `stride` can only be constants.

Here is how you specify a slice for each dimension in a multidimensional array:

```
C and C++:    [slice][slice]...
Fortran:      (slice,slice,...)
```

**Example 1:** A slice declaration of `:::2` for a C or C++ array (with a default lower bound of 0) tells TotalView to display elements with even indices of the array: 0, 2, 4, and so on. However, if this were defined for a Fortran array (with a default lower bound of 1), TotalView displays elements with odd indices of the array: 1, 3, 5, and so on.

**Example 2:** Figure 96 displays a slice of `:::9,:::9`. This definition displays the four corners of a 10-element by 10-element Fortran array.

**Example 3:** You can use a stride to invert the order *and* skip elements. For example, here is a slice that begins with the upper bound of the array and display every other element until it reaches the lower bound of the array: `:::-2`. Thus, using `:::-2` with a Fortran `integer(10)` array tells TotalView to displays the following elements:

```
{10}
{8}
{6}
...
```

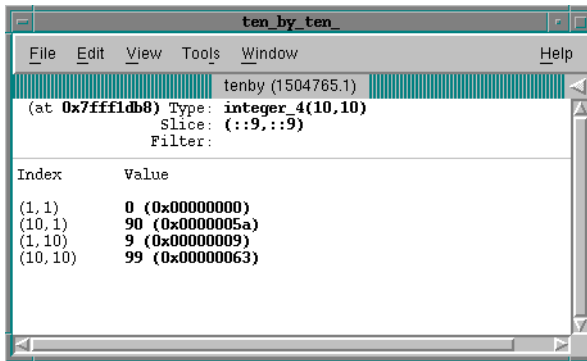


FIGURE 96: Slice Displaying the Four Corners of an Array

**Example 4:** You can simultaneously invert the array's order and limit its extent to display a small section of a large array. The following example specifies a (2:3,7::-1) slice with an `integer*4(-1:5,2:10)` Fortran array:

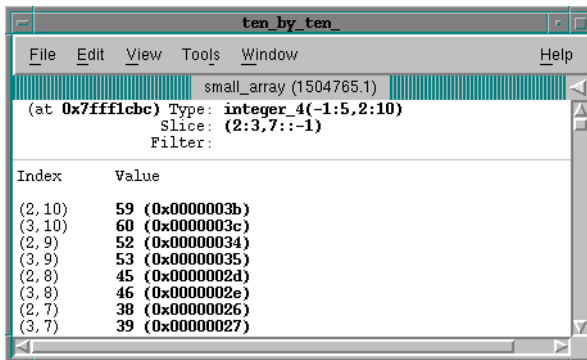


FIGURE 97: Fortran Array with Inverse Order and Limited Extent

After you enter this slice value, TotalView only shows elements in rows 2 and 3 of the array, beginning with column 10 and ending with column 7.

## Using Slices in the Variable Command

When you use the **View > Lookup Variable** command to display a Variable Window, you can include a slice expression as part of the variable name.

Specifically, if you include an array name followed by a set of slice descrip-

tions in the variable dialog box, TotalView initializes the slice field in the Variable Window to the slice descriptions that you specified.

If you include an array name followed by a list of subscripts in the variable dialog box, TotalView interprets the subscripts as a slice description rather than as a request to display an individual value of the array. As a result, you can display different values of the array by changing the slice expression.

For example, suppose that you have a 10-element by 10-element Fortran array named **small\_array**, and you want to display element (5,5). Using the **View > Lookup Variable** command, you enter **small\_array(5,5)**. This sets the initial slice to (5:5,5:5).

You can tell TotalView to display one of the array's values by enclosing the array name and subscripts (that is, the information normally included in a slice expression) within parentheses, such as (**small\_array(5,5)**). In this case, the Variable Window just displays the type and value of the element and does not show its array index. See Figure 98.

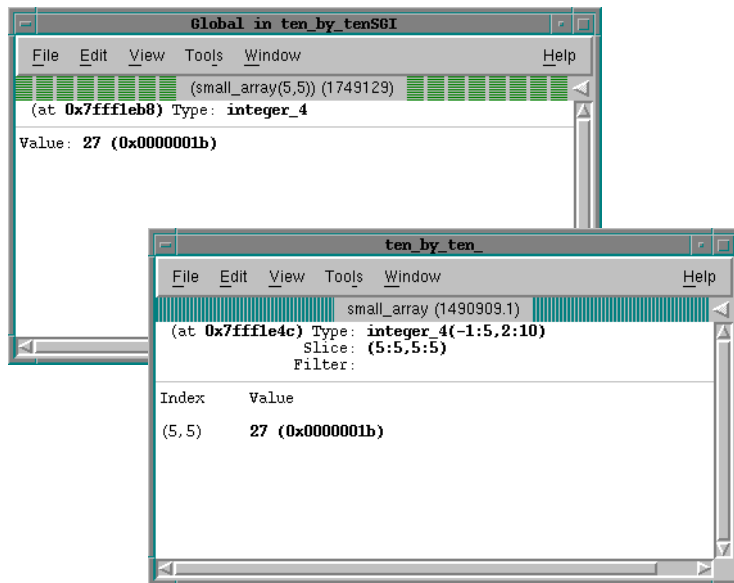


FIGURE 98: Variable Window for *small\_array*

The top left Variable Window in this figure shows the information for `(small_array(5:5))`. The bottom right Variable Window shows the information displayed for `small_array(5:5)`.

## Array Data Filtering

You can filter arrays of type character, integer, or floating-point by specifying a filter expression in the **Filter** field. Your filtering options are:

- Arithmetic comparison to a constant value
- Equal or not equal comparison to IEEE NaNs, INFs, and DENORMs
- Within a range of values, inclusive or exclusive
- General expressions

When an element of an array matches the filter expression, the element is included in the Variable Window display.

You can also sort array elements into an ascending or descending order and display statistical information about the array.

## Filtering by Comparison

Specify arithmetic comparisons to a constant value with the following format:

*operator value*

where *operator* is either a C/C++ or Fortran-style comparison operator, and *value* is a signed or unsigned integer constant, or a floating-point number.

Table 13 lists the comparison operators.

TABLE 13: Array Data Filtering Comparison Operators

Comparison	C/C++ Operator	Fortran Operator
Equal	<code>==</code>	<code>.eq.</code>
Not equal	<code>!=</code>	<code>.ne.</code>
Less than	<code>&lt;</code>	<code>.lt.</code>
Less than or equal	<code>&lt;=</code>	<code>.le.</code>
Greater than	<code>&gt;</code>	<code>.gt.</code>
Greater than or equal	<code>&gt;=</code>	<code>.ge.</code>

The following figure shows an array whose filter is "< 0". This indicates that TotalView should only display array elements whose value is less than 0 (zero).

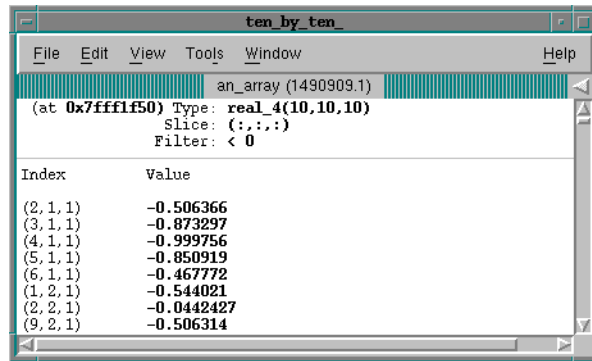


FIGURE 99: Array Data Filtering by Comparison

If the *value* you are using in the comparison is an integer constant, TotalView uses a signed comparison. If you add a **u** or **U** to the constant, TotalView performs an unsigned comparison.

## Filtering for IEEE Values

You can filter IEEE NaN, infinity, or denormalized floating-point values by specifying a filter in the following form:

*operator ieee-tag*

The only comparison operators you can use are *equal* and *not equal*.

The *ieee-tag* represents an encoding of IEEE floating-point values, as explained in the following table:

TABLE 14: Array Data Filtering IEEE Tag Values

IEEE Tag Value	Meaning
\$nan	NaN (Not a number), either Quiet or Signaling
\$nanq	Quiet NaN
\$nans	Signaling NaN
\$inf	Infinity, either Positive or Negative

TABLE 14: Array Data Filtering IEEE Tag Values (cont.)

IEEE Tag Value	Meaning
\$pinf	Positive Infinity
\$ninf	Negative Infinity
\$denorm	Denormalized number, either positive or negative
\$pdenorm	Positive denormalized number
\$ndenorm	Negative denormalized number

Figure 100 shows an example of filtering an array for IEEE values. The bottom left Variable Window shows how TotalView displays the unfiltered array. Notice the INF, -INF, NANQ, and NANS values. Then other two windows show filtered displays. The top right window only shows infinite values. The center window only shows the values of denormalized numbers.

## Filtering by Range of Values

Specify range expressions by using the following format:

[>] *low-value* : [<] *high-value*

where *low-value* specifies the lowest value to include, and *high-value* specifies the highest value to include, separated by a colon. By default, the high and low values are inclusive. If you specify a > before *low-value*, the low value is exclusive. Similarly, a < before the *high-value* makes it exclusive.

The *low-value* and *high-value* must be constants of type integer, unsigned integer, or floating-point. The type of *low-value* must be the same as the type of *high-value*, and *low-value* must be less than *high-value*. If *low-value* and *high-value* are integer constants, they can be immediately followed by **u** or **U**, to force an unsigned comparison. Figure 101 shows a filter applied to an array such that only values equal to or greater than 64 but less than 512 are displayed.

## Array Filter Expressions

The filtering capabilities described in the previous sections are those that are most often used. In some circumstances, you may want to create more general filter expressions. When you create a filter expression, you are creating a Fortran or C Boolean expression that TotalView evaluates for every

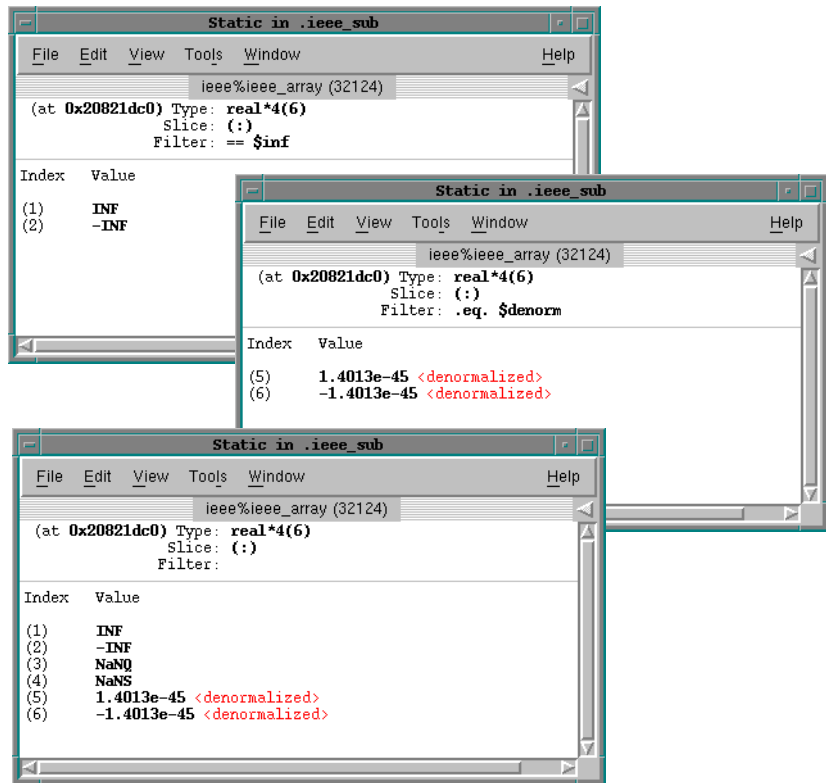


FIGURE 100: Array Data Filtering for IEEE Values

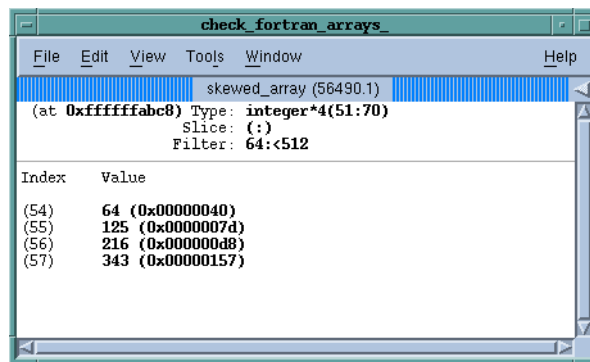


FIGURE 101: Array Data Filtering by Range of Values

element in the array or the array slice. For example, here is an expression that displays all array elements whose contents are greater than 0 and less than 50 or greater than 100 and less than 150.

```
($value > 0 && $value < 50) ||
($value > 100 && $value < 150)
```

As TotalView looks at array elements, it sets the **\$value** special variable to the element's value. It then evaluates your expression. So, if your array had 15 elements, this expression would be evaluated 15 times.

Notice also the use of the **&&** and **||** operators to joint parts of the Boolean expression together. You can use any of TotalView's standard operators. And, the way in which TotalView computes the results of an expression is identical to the way it computes values at an evaluation point. For more information, see "Defining Evaluation Points" on page 216.

**NOTE** You cannot use any of the IEEE tag values described in "Filtering for IEEE Values" on page 189 in these kind of expressions.

## Filter Comparisons

TotalView lets you filter array information in a variety of ways—and these ways can overlap. For example, the following two filters produce the same result:

```
> 100
$value > 100
```

Similarly, you obtain the same results with either of the following:

```
>0:<100
$value > 0 && $value < 100
```

In both of these, the first method is easier to type than the second. In general, you would use the second method when you are creating more complicated expressions.



## Filtering Array Data

The procedure for filtering an array is quite simple: select the **Filter** field, enter the array filter expression, and then press Return.

TotalView updates the Variable Window to exclude only the elements that do not match the filter expression.

TotalView applies the filter expression to each element of the array after any array slice is applied. TotalView displays the element if its value matches the filter expression.

If necessary, TotalView converts the array element before evaluating the filter expression. The following conversion rules apply:

- If the filter operand or array element type is floating-point, TotalView converts it to a double-precision floating-point value. Extended-precision values are truncated to double precision. Converting integer or unsigned integer values to double-precision values may result in a loss of precision. Unsigned integer values are converted to non-negative double-precision values.
- If the filter operand or the array element is an unsigned integer, TotalView converts the values to an unsigned 64-bit integer.
- If both the filter operand and array element are of type integer, TotalView converts the values to type 64-bit integer.

These conversions modify a copy of the array's elements—they never alter the actual array elements.

To stop filtering an array, delete the contents of the **Filter** field in the Variable Window and press Return. TotalView will then update the Variable Window so that it includes all elements.

## Sorting Array Data

TotalView lets you sort the displayed array data into ascending or descending order. (It does not, of course, sort the actual data.)

If you select the Variable Window's **View > Sort > Ascending**, TotalView places all of the array's elements in ascending order. (See Figure 102 for an example.)

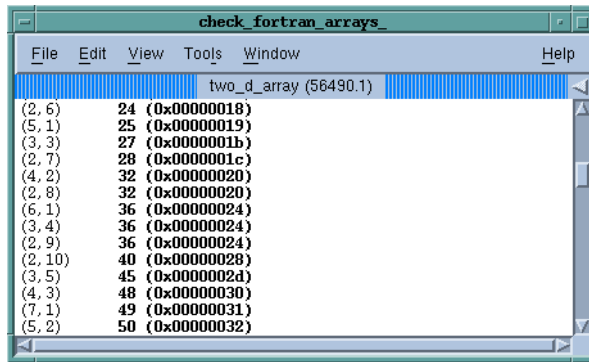


FIGURE 102: Sorted Variable Window

As you would expect, **View > Sort > Descending** places array elements into descending order. The **View > Sort > None** command returns the array to its original order.

The sort commands only manipulate the displayed elements. This means that if you limit the number of elements by defining a slice or a filter, TotalView only sorts the result of the filtering and slicing operations.

## Array Statistics

The **Tools > Statistics** command displays a window containing information about your array. Figure 103 shows an example.

If you have added a filter or a slice, these statistics only describe the information that is being displayed; the statistics do not describe the entire unfiltered array.

The statistics TotalView displays are as follows:

### ■ Checksum

A checksum value for the array elements.

### ■ Count

The total number of displayed array values. If you are displaying a floating-point array, this number does not include NaN or Infinity values.

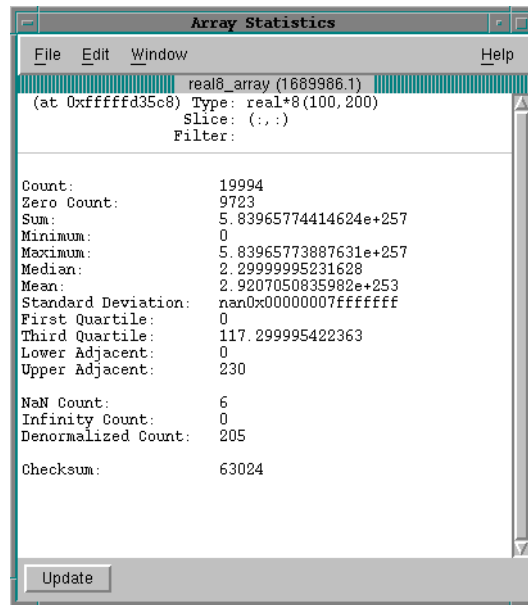


FIGURE 103: Array Statistics Window

### ■ Denormalized Count

A count of the number of denormalized values found in a floating-point array. This includes both negative and positive denormalized values as defined in the IEEE floating-point standard. Unlike other floating-point statistics, these elements participate in the statistical calculations.

### ■ Infinity Count

A count of the number of infinity values found in a floating-point array. This includes both negative and positive infinity as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

### ■ Lower Adjacent

This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first quartile value less 1.5 times the difference between the first and third quartiles.

### ■ Maximum

The largest array value.

**■ Mean**

The average value of array elements.

**■ Median**

The middle value. Half of the array's values are less than the median and half are greater than the median.

**■ Minimum**

The smallest array value.

**■ NaN Count**

A count of the number of NaN values found in a floating-point array. This includes both signaling and quiet NaNs as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

**■ Quartiles, First and Third**

Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the fourth quartile value means that 75% of the array's values are less than this value and 25% are greater.

**■ Standard Deviation**

The standard deviation for the array's values.

**■ Sum**

The sum of all of the displayed array's values.

**■ Upper Adjacent**

This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus 1.5 times the difference between the first and third quartiles.

**■ Zero Count**

The number of elements whose value is 0.

## Displaying a Variable in All Processes or Threads

When you are debugging a parallel program that is running many instances of the same executable, you usually need to view or update the value of a variable in all of the processes or threads at once.

To display the value of a variable in all of the processes in a parallel program, first bring up a Variable Window displaying the value of a variable in one of the processes. You can now use these commands:

- **View > Laminated > Process** displays the value of the variable in all of the processes.
- **View > Laminated > Thread** displays the value of a variable in all threads within a single process.

**NOTE** You cannot simultaneously laminate across processes and threads in the same Variable Window.

The Variable Window switches to "laminated" mode, and displays the value of the variable in each process or thread. Figure 104 shows a simple, scalar variable in each of the processes in some OpenMP programs. Notice that the first six have a variable in a matching stack frame. The corresponding variable cannot be found for the seventh thread.

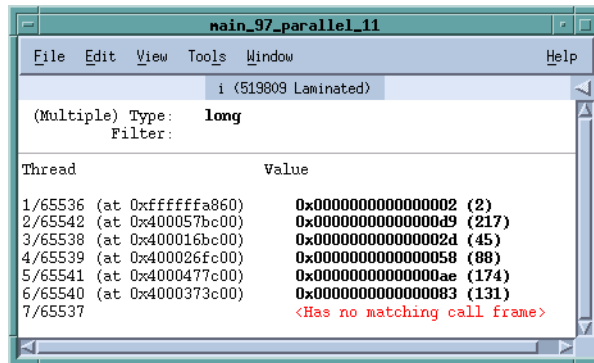


FIGURE 104: Laminated Scalar Variable

If you decide that you no longer want the pane to be laminated, use the **View > Laminated > None** command to delaminate it.

When looking for a matching stack frame, TotalView matches frames starting from the top frame, and considers calls from different memory or stack locations to be different calls. For example, the following definition of **recurse** contains two additional calls to **recurse**. Each of these generate nonmatching stack frames.

```
int recurse (int i, int depth)
{
    if (i == 0)
        return depth;
    if (i&1)
        recurse (i-1, depth+1);
    else
        recurse (i-3, depth+1);
}
```

If the variables are at different addresses in the different processes or threads, the address field at the top of the pane displays **(Multiple)** and the unique addresses are displayed with each data item, as was shown in Figure 104.

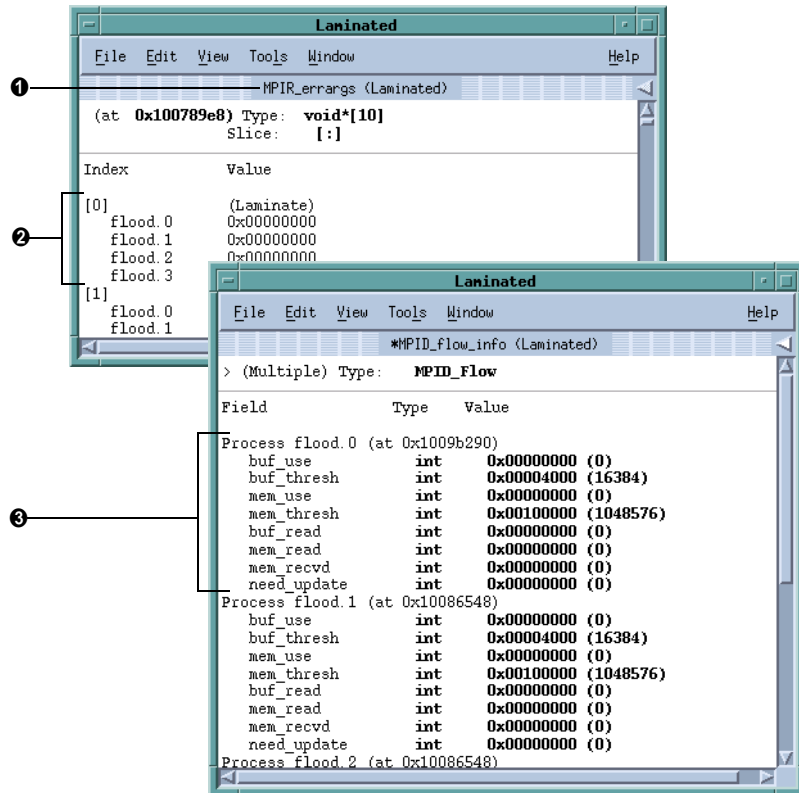
TotalView also allows you to laminate arrays and structures. When you laminate an array, each element in the array is displayed across all processors. As with a normal Variable Window, you can use a slice to select elements to be displayed. Figure 105 shows an example of a laminated array and a laminated structure. You can also laminate an array of structures.

## Diving in a Laminated Pane

You can dive through pointers in a laminated Variable Window, and the dive will apply to the associated pointer in each process or thread.

## Editing a Laminated Variable

If you edit a value in a laminated Variable Window, TotalView asks if it should apply this change to all of the processes or threads or only the one in which you made a change. This is also an easy way to update a variable in all of processes



- ❶ Laminated array
- ❷ Element [0] for each of the processes
- ❸ Structure elements for one process

FIGURE 105: Laminated Array and Structure

## Visualizing Array Data

The TotalView Visualizer works with TotalView to create graphic images of array data in your programs. This lets you see your data in one glance and quickly find problems with it as you debug your programs.

You can execute the Visualizer from within TotalView or you can run it from the command line to visualize data dumped to a file in a previous TotalView session.

For information about running the TotalView Visualizer, see Chapter 10, “*Visualizing Data*” on page 247.

### Visualizing a Laminated Variable Window

You can export data from a laminated Variable Window to the Visualizer by using the **Tools > Visualize** command. When visualizing laminated data, the process (or thread) index is the first axis of the visualization. This means that you must use one fewer data dimension than you normally would. If you do not want the process/thread axis to be significant, you can use a normal Variable Window since all of the data must necessarily be in one process.



# Setting Action Points

This chapter explains how to use action points. TotalView supports four kinds of action points: breakpoints, barrier breakpoints, evaluation points, and watchpoints. A *breakpoint* stops execution of processes and threads that reach it. A *barrier* breakpoint holds each thread and process that reaches it until all threads and processes from the group reach it. An *evaluation point* causes a code fragment to execute when it is reached. A *watchpoint* lets you monitor a location in memory and stop execution when the value stored in memory changes.

Topics in this chapter are:

- Action Points Overview
- Setting Breakpoints and Barriers
- Defining Evaluation Points
- Using Watchpoints
- Saving Action Points to a File
- Evaluating Expressions
- Writing Code Fragments

## Action Points Overview

Action points allow you to specify an action that will be performed when a thread or process reaches a source line or machine instruction in your program. Here are the different kinds of action points that you can use:

### ■ Breakpoints

When a thread encounters a breakpoint, it stops at the breakpoint along with the other threads in the process. You can also arrange for other related processes to stop when a breakpoint is hit.

Breakpoints are the simplest action point.

### ■ Barrier breakpoints

Barrier breakpoints are similar to simple breakpoints, differing in that they are used to synchronize a group of processes or threads. Barrier breakpoints work together with the TotalView hold and release feature. TotalView supports thread barrier and process barrier breakpoints.

### ■ Evaluation points

An evaluation point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an evaluation point, it executes this code. You can use evaluation points in several different ways, including conditional breakpoints, thread-specific breakpoints, count-down breakpoints, and patching code fragments into and out of your program.


### ■ Watchpoints

A watchpoint tells TotalView that it should either stop the thread so that you can interact with your program (unconditional watchpoint) or evaluate an expression (conditional watchpoint).

All action points share some common properties. They:

- Can be enabled or disabled independently. A disabled action still exists; however, when your program reaches a disabled point, the program continues executing.
- Can be shared across multiple processes, or set in individual processes.
- Apply to the process, so in a multithreaded process, the action point applies to all of the threads.
- Are assigned unique action point ID numbers. They appear in several places, including the Root Window, the Action Points Pane of the Process Window, and the **Action Point > Properties** dialog box.

Each type of action point has a unique symbol. Figure 106 shows examples of some enabled and disabled action points:

The  icon indicates that there are one or more assembler-level action points associated with the source line.

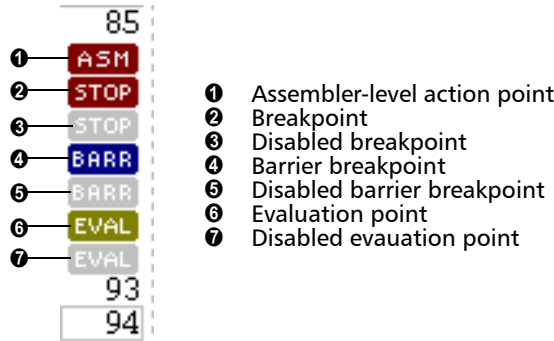


Figure 106: Action Point Symbols

## Setting Breakpoints and Barriers

TotalView has several options for setting breakpoints. You can set:

- Source-level breakpoints
- Machine-level breakpoints
- Breakpoints that are shared among all processes in multiprocess programs

You can also control whether or not TotalView stops all processes in the control group when a single member reaches a breakpoint.

### Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a boxed line number in the tag field of the Process Window. (A boxed line number indicates that the line is associated with executable code.) A **STOP** icon lets you know that a breakpoint is set immediately before the source statement.

You can also set a breakpoint while a process is running by selecting a boxed line number in the tag field of the Process Window. If you set a breakpoint while the process is running, TotalView temporarily stops the process so it can insert the breakpoint. After the breakpoint is set, the process resumes executing.

## Selecting Ambiguous Source Lines

If you are using C++ templates, a single source line could generate multiple function instances. If you attempt to set a source-level breakpoint by selecting a line number in a function template, and that line number has more than one instantiation, TotalView will prompt you with an **Ambiguous Line** dialog box, as shown in Figure 107.

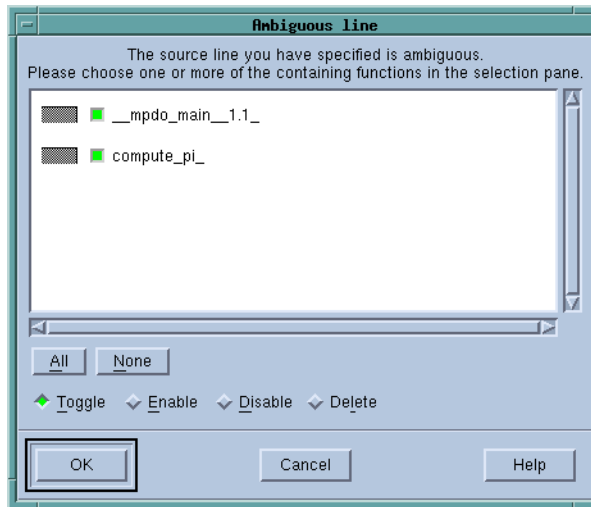


Figure 107: Ambiguous Line Dialog Box

Here is how you use this dialog box:

- 1 Select functions by checking:
  - All**, which selects all functions.
  - None**, which deselects all functions.
  - Individual check boxes, which lets you select and deselect functions.
- 2 Select one of the following:
  - Toggle**, which changes the state of the action points.
  - Enable**, which enables the action points, or creates breakpoints or barrier breakpoints for any that did not already exist.
  - Disable**, which disables the action point.
  - Delete**, which deletes breakpoints or barrier breakpoints, and disables others.
- 3 Select the **OK** button or press Return to perform the action.

## Toggling Breakpoints at Locations

You can toggle a breakpoint at a specific function or source-line number without having to first find the function or source line in the Source Pane by using the following procedure:

- 1 Invoke the **Action Point > At Location** command. The **At Location** dialog box appears (as shown in Figure 108).

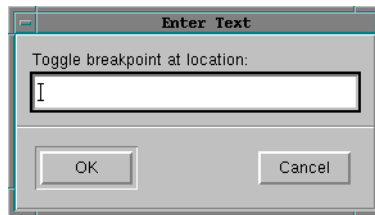


Figure 108: **Action Point > At Location** Dialog Box

- 2 Type the name of the function or a source-line number.  
Entering a function name tells TotalView to toggle the breakpoint at the function's first executable source line. Entering a source-line number toggles the breakpoint at the source line in the current source file.
- 3 Select **OK**.

## Ambiguous Locations

If you enter an ambiguous function name with the **Action Point > At Location** command, TotalView displays one of its **Ambiguous Function Name** dialog boxes. See Figure 109.

The procedure for resolving ambiguous function names is similar to the procedure described in "Selecting Ambiguous Source Lines" on page 204.

## Displaying and Controlling Action Points

The **Action Point > Properties** dialog box lets you set and control an action point. Figure 110 on page 206 shows this dialog box. It also allows you to change an action point's type among one of the three kinds: breakpoint, barrier point, and evaluation point. This box also lets you define what will happen to other threads and processes when execution reaches a breakpoint or barrier point.

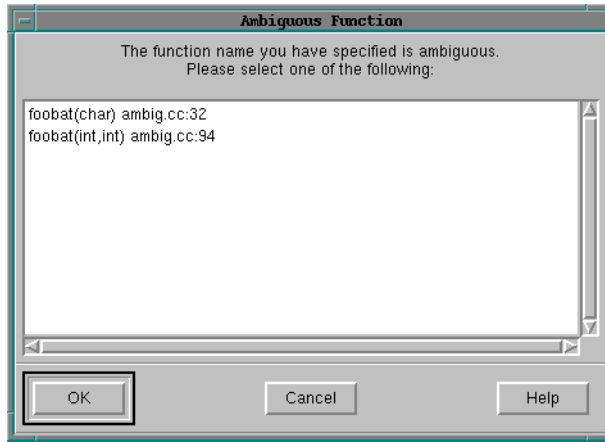


Figure 109: Ambiguous Function Dialog Box

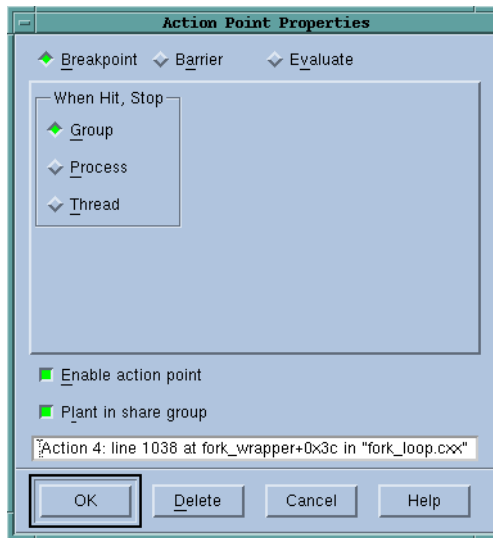


Figure 110: Action Point &gt; Properties Dialog Box

The following sections explain how you can control action points by using the Process Window and the **Properties** dialog box.

## Disabling

TotalView can retain an action point's definition but ignore the action point during execution. That is, disabling an action point does not remove it. TotalView remembers that an action point exists for the line, but ignores it as long as it is disabled.

You can disable an action point by:

- Deselecting **Enable action point** in the **Properties** dialog box.
- Selecting the **STOP** or **BARR** sign in the Action Points Pane.

## Deleting

You can permanently remove an action point by selecting the **STOP** or **BARR** sign in the tag field or selecting the **Delete** button in the **Action Point > Properties** dialog box.

To delete all breakpoints and barrier points, select the **Action Point > Delete All** command.

## Enabling

You can activate an action point that was previously disabled by selecting the dimmed **STOP**, **BARR**, or **EVAL** sign in the process or Action Points Pane, or by selecting **Enable action point** in the **Properties** dialog.

## Suppressing

You can tell TotalView to ignore action points and prevent the creation of additional action points by selecting the **Action Point > Suppress All** command.

When you suppress action points, you disable them. If action points are suppressed, you cannot update existing action points or create new ones.

You can make previously suppressed action points active and allow the creation of new ones by reselecting the **Action Point > Suppress All** command.

## Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code or source interleaved with assembler. (Refer to “*Examining Source and Assembler Code*” on page 129 for information.) You can now select the tag field opposite an instruction. The tag field must contain a gridset (:::)—the gridset indicates the line is the beginning of a machine instruction. Since instruction sets on some platforms support variable-length instructions, you may see multiple lines associated with a single gridset. The **STOP** icon appears, indicating that the breakpoint occurs before the instruction executes.

When the Source Pane displays source interleaved with assembler, source statements are treated as if they were comments: they are not treated as executable statements. (This is shown in Figure 111.) Because they are treated as comments, you cannot set breakpoints on them. If you set a breakpoint on the first instruction after a source statement, however, you are actually creating a source-level breakpoint.

```


:::      0x10005ae8:      0x0320f809  jalr    t9
:::      0x10005aec:      0x00000000  nop
934      .
935      printf ("Parent %d forked new child %d\n", (int)(getpid()),
STOP      0x10005af0:      0xdf998110  ld      t9, -0x7ef0(gp)
:::      0x10005af4:      0x0320f809  jalr    t9
:::      0x10005af8:      0x00000000  nop
:::      0x10005afc:      0x00402825  or      a1, v0, zero
:::      0x10005b00:      0xffa50070  sd      a1, 0x70(sp)

```

Figure 111: Breakpoint at Assembler Instruction

If you set machine-level breakpoints on one or more instructions that are part of a single source line and then display source code in the Source Pane, TotalView displays an **ASM** icon (see Figure 106) on the line number. To see the specific breakpoints, you must display assembler or assembler interleaved with source code.

When a process reaches a breakpoint, TotalView:

- Suspends the process.
- Displays the PC arrow icon (  ) over the stop sign to indicate that the PC currently points to the breakpoint. (See Figure 112.)
- Displays **At Breakpoint** in the Process Window title bar and other windows.
- Updates the Stack Trace and Stack Frame Panes and Variable Windows.



```

80      do 40 i = 1, 500
81          denorms(i) = x'00000001'
82      continue
40      do 42 i = 500, 1000
83  STOP  denorms(i) = x'80000001'
84      continue
85  42      local_var=100
86      ieee_array(1) = x'7f800000'      | infinity
87      ieee_array(2) = x'ff800000'      | -infinity
88      ieee_array(3) = x'7fc00001'      | NANQ
89      ieee_array(4) = x'7f800001'      | NANS
90      ieee_array(5) = x'00000001'      | positive denormalized number
91      ieee_array(6) = x'80000001'      | negative denormalized number
92

```

Figure 112: Assembler and Source Interleaved

## Breakpoints for Multiple Processes

In multiprocess programs, you can set breakpoints in the parent process and child processes before you start the program and at any time during its execution. To do this, use the **Action Point > Properties** dialog box, as shown in Figure 113. This dialog box provides the following controls for setting breakpoints:

### ■ When Hit, Stop

When your thread hits a breakpoint, TotalView can also stop the thread's control group or the process in which it is running.

### ■ Plant in share group

If selected, TotalView enables the breakpoint in all members of this thread's share group at the same time. If this is not selected, you must individually enable and disable breakpoints in each share group member.

You can control the default setting of these check boxes using X resources or command-line options. See Figure 113.

The action point ID and other information are displayed at the top of the dialog box.

You can preset many of the properties in this dialog box by using TotalView preferences, as shown in Figure 114 on page 210.

You can find additional information about this dialog box within the Help.

In addition to the controls in the **Properties** dialog box, you can place an expression in the expression box to control the behavior of control group members and share group members. Refer to "Writing Code Fragments" on page 234 for more information.

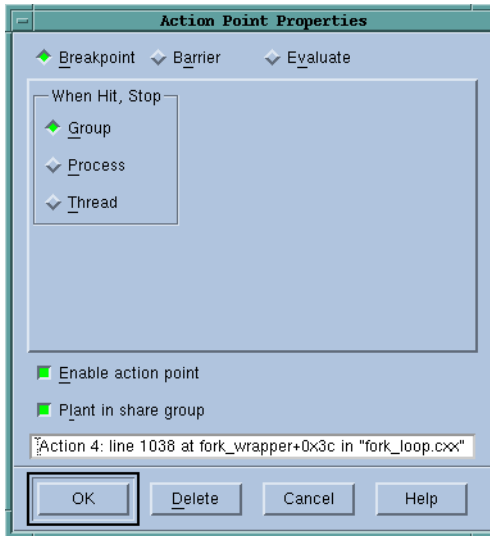


Figure 113: Action Point &gt; Properties Dialog Box

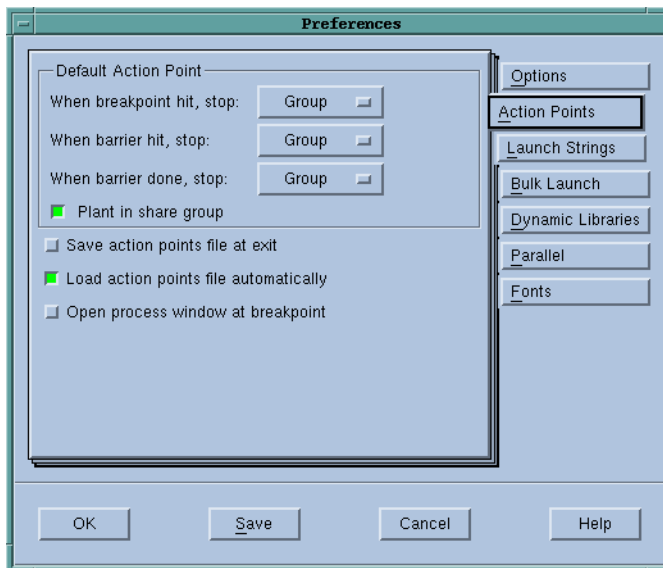


Figure 114: File &gt; Properties: Action Points Page

If you are trying to synchronize your program's threads, you will want to set a barrier point. For more information, see "Barrier Breakpoints" on page 212.

## Breakpoint When Using fork()/execve()

You must link with the **dbfork** library to debug programs that call **fork()** and **execve()**. See "Compiling Programs" on page 15.

### Processes That Call fork()

By default, TotalView places breakpoints in all processes in the share group. When any of these processes reach a breakpoint, TotalView stops all processes in the control group. Said in a different way, TotalView stops the control group containing the share group. This means that TotalView will be stopping more than one share group. (For more information, see "Examining Groups" on page 135.) To override these defaults:

- 1 Dive into the tag field to display the **Action Point > Properties** dialog box.
- 2 Deselect the **Plant in share group** checkbox and make sure that the **Group** radio button is selected.

### Processes That Call execve()

Breakpoints that are shared by a parent and children with the same executable do not apply to children with different executables. To set the breakpoints for children that call **execve()**:

- 1 Set the breakpoints and breakpoint options desired in the parent and the children that do not call **execve()**.
- 2 Start the multiprocess program by displaying the **Group > Go** command. When the first child calls **execve()**, TotalView displays the following message:  

Process name has exec'd *name*.  
Do you want to stop it now?
- 3 Answer **Yes**. TotalView opens a Process Window for the process. (If you answer **No**, TotalView will not allow you to set breakpoints.)
- 4 Set breakpoints for the process. After you set breakpoints for the first child using this executable, TotalView does not prompt when other children call **execve()** to use it. Therefore, if you do not want to

share the breakpoints among other children using the same executable, dive into the breakpoints and set the breakpoint options.

**5** Select the **Group > Go** command.

### Example: Multiprocess Breakpoint

The following partial program illustrates the different points at which you can set breakpoints in a multiprocess program:

```

1  pid = fork();
2  if (pid == -1)
3      error ("fork failed");
4  else if (pid == 0)
5      children_play();
6  else
7      parents_work();

```

The following table shows the results of setting a breakpoint at different places.

Table 15: Setting Breakpoints in Multiprocess Programs

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes (if the child process was successfully created).
3	Stops the parent process if <b>fork()</b> failed.
5	Stops the child process.
7	Stops the parent process.

### Barrier Breakpoints

A barrier breakpoint is similar to a simple breakpoint, differing in that it holds processes and threads that reach the barrier point. Other processes and threads continue to run. TotalView holds each until all the processes or threads defined in the barrier point reach this same place. When the last process or thread reaches a barrier point, TotalView releases all the held processes or threads.

## Barrier Breakpoint States

Processes and threads at a barrier point are held or stopped, as follows:

<b>Held</b>	A process or thread that is <i>held</i> cannot resume execution until all the processes or threads in its group are at the barrier, or until you manually release it. The various <i>go</i> and <i>step</i> commands from the <b>Group</b> , <b>Process</b> , and <b>Thread</b> menus have no effect on held processes.
<b>Stopped</b>	When all processes in the group reach a barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you tell them to resume executing.

You can manually release held processes and threads with the **Hold** and **Release** commands contained within the **Group**, **Process**, and **Thread** menus. When you manually release a process, the *go* and *step* commands become available again.

You can reuse the **Hold** command to again toggle the hold state of the process or thread. See “*Holding and Releasing Processes and Threads*” on page 134 for more information.

## Setting a Barrier Breakpoint

You can set a barrier breakpoint by using the **Action Point > Set Barrier** command or from the **Action Point > Properties** dialog box which is shown in Figure 115. (Right-clicking on the line also allows you to set a barrier.)

Barrier breakpoints are most often used to synchronize a set of threads. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread reaching the barrier from responding to resume commands (for example, *step*, *next*, or *go*) until all threads in the affected set arrive at the barrier. When all threads reach the barrier, TotalView considers the barrier to be *satisfied* and releases them. *They are just released; they are not continued.* That is, they are left stopped at the barrier. If you now continue the process, those threads stopped at the barrier also run. This is in addition to any other threads that were not affected with the barrier.

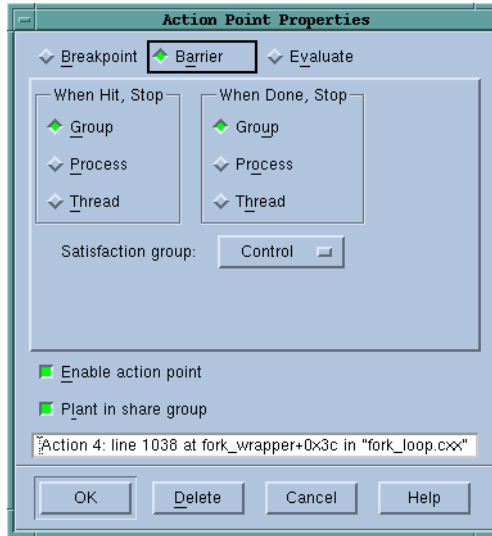


Figure 115: Action Point > Properties Dialog Box

If a process is stopped and then continued, the held threads, including the ones waiting at an unsatisfied barrier, do not run. Only unheld threads run.

The **When Hit, Stop** radio buttons indicate what other threads TotalView should stop when execution reaches the breakpoint, as follows:

Scope	TotalView will:
Group	Stop all threads in the current thread's control group.
Process	Stop all threads in the current thread's process.
Thread	Only stop this thread.

After all processes or threads reach the barrier, TotalView releases all held threads. (*Released* means that these threads and processes can now run.)

The **When Done, Stop** radio buttons tell TotalView what else it should stop, as follows.

Scope	TotalView will:
Group	Stop all threads in the current thread's control group.
Process	Stop all threads in the current thread's process.
Thread	Only stop this thread.

For even more control over what TotalView will stop, you can indicate a *satisfaction set*. This set indicates which threads must be held before TotalView will release the group of threads. That is, the barrier is *satisfied* when TotalView has held all of the indicated threads. Use the **Satisfaction group** items to tell TotalView that the satisfaction set consists of all threads in the current thread's **Share**, **Workers**, or **Lockstep** group.

When you set a barrier point, TotalView places it in every process within the share group.

If you run one of the processes or threads in a group and it hits a barrier point, you will see an **H** next to the process or thread name in the Root Window and the word [**Held**] in the title bar in the main Process Window. Barrier points are always shared.

If you create a barrier and all the processes threads are already at that location, TotalView does not hold any of them. However, if you create a barrier and all of the processes and threads are not at that location, TotalView holds the ones that already there, and does not hold the ones that are not there yet.

## Releasing Processes from Barrier Points

TotalView automatically releases processes and threads from a barrier point when they hit that barrier point and all other processes or threads in the group are already held at it.

## Deleting a Barrier Point

You can delete a barrier point in two ways:

- Using the **Action Point > Properties** dialog box.
- Clicking on the  icon in the line number area.

## Changes When Setting and Clearing a Barrier Point

Setting a barrier point at the current PC for a *stopped* process or thread holds the process there. If, however, all other processes or threads affected by the barrier point are at the same PC, TotalView does not hold them. Instead, TotalView treats the barrier point as if it was an ordinary breakpoint.

All processes and threads that are held and which have threads at the barrier point are released when you clear the barrier point. They remain stopped but are no longer held. You can clear the barrier breakpoint in the **Action Point > Properties** dialog box by clicking on **Clear** at the bottom of the dialog box.

## Defining Evaluation Points

TotalView lets you define *evaluation points*. These are action points at which you have added a code fragment that TotalView will execute. You can write the code fragment in C, Fortran, or assembler.

**NOTE** Assembler support is currently available on the Compaq Tru64 UNIX, IBM AIX, and SGI IRIX operating systems. Compiled expressions must be enabled to use assembler constructs.

Here are some of the ways you can use evaluation points:

- A fragment can also include instructions that stop a process and its relatives. If the code fragment can make a decision whether it should stop execution, it is called a *conditional breakpoint*.
- You can also use evaluation points to test potential fixes for your program.
- You can set values of your program's variables.
- You can automatically send data to the Visualizer. This can produce animated displays of the changes in your program's data.

You can set an evaluation point at any source line that generates executable code (marked with a boxed line number in the tag field). You can also define evaluation points on machine-level instructions. Stated in a different way, if you can set a breakpoint, you can change it into an evaluation point.

At each evaluation point, the code fragment in the evaluation point is executed before the code on that line. Typically, your program will then execute the program instruction at which you had set the evaluation point. But your code fragment can modify this behavior:



- It can include a branching instruction (such as **goto** in C or Fortran). The instruction can transfer control to a different point in the target program, enabling you to test program patches.
- It can contain an intrinsic statement. These special TotalView statements let you stop execution, create barriers, and countdown breakpoints. For more information on these statements, refer to Table 19 “Built-In Statements Used in Expressions” on page 236.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and branch to places in your program.

For complete information on what you can include in code fragments, refer to “Writing Code Fragments” on page 234.

Evaluation points only modify the processes being debugged—they do not modify the source program or create a permanent patch in the executable. If you save a program’s evaluation points, however, TotalView reapplies them whenever you start a debugging session for that program. To save your evaluation points, refer to “Saving Action Points to a File” on page 232.

**NOTE** You should stop a process before setting an evaluation point. This ensures that the evaluation point is set in a stable context in the program.

## Setting Evaluation Points

To set an evaluation point:

- 1 Display the **Action Point > Properties** dialog box. You can do this, for example, by right-clicking on a **STOP** icon and selecting **Properties** or by selecting a line and then invoking the command from the menu bar.
- 2 Select the **Evaluate** button.
- 3 Select the button (if it is not already selected) for the language in which you will code the fragment.
- 4 Type the code fragment. For information on supported C, Fortran, and assembler language constructs, refer to “Writing Code Fragments” on page 234.

- 5 For multiprocess programs, decide whether to share the evaluation point among all processes in the program's share group. By default, TotalView selects the **Plant in share group** check box for multiprocess programs, but you can override this by deselecting it.
- 6 Select the **OK** button to confirm your changes. If the code fragment has an error, TotalView displays an error message. Otherwise, it processes the code, closes the dialog box, and places an **EVAL** icon in the tag field.

## Creating Conditional Breakpoint Examples

Here are some examples of conditional breakpoints code:

- To define a breakpoint that is reached whenever variable *i* is greater than 20 but less than 25:

```
if (i > 20 && i < 25)
    $stop;
```

- To define a breakpoint that will stop execution every 10th time that TotalView executes the **\$count** statement:

```
$count 10
```

- To define a breakpoint with a more complex expression, consider:

```
$count i * 2
```

When the variable *i* equals 4, the process stops the 8th time it executes the **\$count** statement. After the process stops, the expression is reevaluated. If *i* now equals 5, the next stop occurs after the process executes the **\$count** statement 10 more times.

For complete descriptions of the **\$stop** and **\$count** statements, refer to "Built-In Statements" on page 236.

## Patching Programs

You can use expressions in evaluation points to patch your code if you use the **goto** (C) and **GOTO** (Fortran) statements to jump to a different program location. This lets you:

- Branch around code that you do not want your program to execute.
- Add new pieces of code.

In many cases, correcting an error means that you will do both operations: you patch out incorrect lines and patch in corrections.

## Conditionally Patching Out Code

The following example contains a logic error where the program dereferences a null pointer:

```

1  int check_for_error (int *error_ptr)
2  {
3      *error_ptr = global_error;
4      global_error = 0;
5      return (global_error != 0);
6  }
```

The error occurs because the routine calling this function assumes that the value of **error\_ptr** can be 0. The **check\_for\_error()** function, however, assumes that **error\_ptr** is not null, which means that line 3 can dereference a null pointer.

You can correct this error by setting an evaluation point on line 3 and entering:

```
    if (error_ptr == 0) goto 4;
```

If the value of **error\_ptr** is null, line 3 is not executed.

## Patching in a Function Call

Instead of routing around the problem, you could patch in a **printf()** statement that displays the value of the **global\_error** variable created in the preceding program. You would set an evaluation point on line 4 and enter:

```
    printf ("global_error is %d\n", global_error);
```

This code fragment is executed before the code on line 4; that is, it is executed before **global\_error** is set to 0.

## Correcting Code

The next example contains a coding error: the function returns the maximum value instead of the minimum value:

```

1  int minimum (int a, int b)
2  {
3      int result;      /* Return the minimum */
4      if (a < b)
5          result = b;
6      else
```

```

7          result = a;
8      return (result);
9  }
```

You can correct this error by adding the following code to an evaluation point at line 4:

```
if (a < b) goto 7; else goto 5;
```

This effectively replaces the **if** statement on line 4 with the statement entered at the evaluation point.

## Interpreted vs. Compiled Expressions

On most platforms, TotalView executes interpreted expressions. TotalView can also execute compiled expressions on the Compaq Tru64 UNIX, IBM AIX, and SGI IRIX platforms. On Compaq Tru64 UNIX and IBM AIX platforms, compiled expressions are enabled by default.

You can enable or disable compiled expressions by using X resources or command-line options. Refer to **totalview\*compileExpressions** on page 277. See Appendix B “*Operating Systems*” on page 321 to find out how TotalView handles expressions on specific platforms.

**NOTE** Using any of the following intrinsics means that TotalView interprets the evaluation point instead of compiling it: `$visualize`, `$nid`, `$clid`, `$processduid`, `$duid`, `$tid`, and `$systid`. In addition, `$pid` forces interpretation on AIX.

## Interpreted Expressions

TotalView sets a breakpoint in your code and executes the evaluation point. Since TotalView is executing the expression, interpreted expressions run slower (and possibly much slower) than compiled expressions. With multiprocess programs, interpreted expressions run even more slowly because processes may need to wait for TotalView to execute the expression.

When you are debugging remote programs, interpreted expressions always run more slowly because TotalView on the host, not the TotalView debugger server (**tvdsrvr**) on the client, executes the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView debugger process on the host machine to evaluate one inter-

interpreted expression. In contrast, if TotalView compiles the expression, it is evaluated on each remote process.

If the expression contains **\$stop**, TotalView stops evaluating the expression and stops the process when it executes the **\$stop** intrinsic.

**NOTE** Whenever a thread hits an interpreted patch point, TotalView stops execution. This means that TotalView will create a new set of lockstep groups. Consequently, if goal threads contain interpreted patches, the results are unpredictable.

## Compiled Expressions

TotalView compiles, links, and patches expressions into the target process. by replacing an instruction with a “branch out” instruction, relocating the original instruction, and appending the expression. This means that the target thread executes this code with the result being that evaluation points and conditional breakpoints execute very quickly. (Note that conditional watchpoints are always interpreted.) And, more importantly, this code does not need to communicate with the TotalView host process until it needs to.

If the expression executes a **\$stop** intrinsic, TotalView stops executing the compiled expression. At this time, you can single-step through it and continue executing the expression as you would the rest of your code. See Figure 116.

If you plan to use compiled expressions, you may need to think about allocating patch space. For more information, see “*Allocating Patch Space for Compiled Expressions*” on page 221.

## Allocating Patch Space for Compiled Expressions

TotalView must allocate or find space in your program to hold the code fragments generated by compiled expressions. Since this patch space is part of your program’s address space, the location, size, and allocation scheme that TotalView uses may conflict with your program. As a result, you may need to change how TotalView allocates this space.

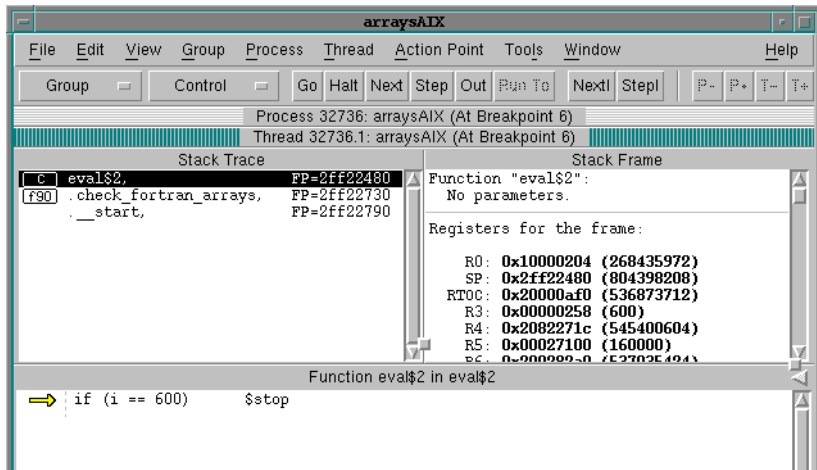


Figure 116: Stopped Execution of Compiled Expressions

You can choose one of the following patch space allocation schemes:

- **Dynamic patch space allocation:** Tells TotalView to find the space for the code fragment dynamically.
- **Static patch space allocation:** Tells TotalView to use a statically allocated area of memory.

## Dynamic Patch Space Allocation

*Dynamic patch space allocation* means that TotalView dynamically allocates patch space for code fragments. If you do not specify the size and location for the patch space, TotalView allocates 1 MB. TotalView creates this space using system calls.

TotalView allocates memory for read, write, and execute access within the following addresses:

Table 16: Dynamic Patch Space Allocation Default Addresses

Platform	Address range
Compaq Tru64 UNIX	0xFFFFF00000 – 0xFFFFFFFF
IBM AIX	0xCFF00000 – 0xCFFFFFFF
SGI IRIX (–n32)	0x4FF00000 – 0x4FFFFFFF
SGI IRIX (–64)	0x8FF00000 – 0x8FFFFFFF

**NOTE** You can only allocate dynamic patch space for these machines.

If the default address range conflicts with your program, or you would like to change the size of the dynamically allocated patch space, you can specify the dynamically allocated:

- Patch space base address by using the `-patch_area_base` command-line option or the X resource `totalview*patchAreaAddress` on page 280.
- Patch space length by using the `-patch_area_length` command-line option or the X resource `totalview*patchAreaLength` on page 280.

## Static Patch Space Allocation

TotalView can statically allocate patch space if you add a specially named array to your program. You can then specify the size of the patch space. When TotalView needs to use patch space, it uses this space created for this array.

To include a 1 MB statically allocated patch space in your program, add the `TVDB_patch_base_address` data object in a C module. Because this object must be 8-byte aligned, declare it as an array of doubles. For example:

```
/* 1 megabyte == size TV expects */
#define PATCH_LEN 0x100000
double TVDB_patch_base_address [PATCH_LEN / sizeof(double)]
```

If you need to use a static patch space size that differs from the 1 MB default, you must use assembler language. Table 17 shows sample assembler code for three platforms that support compiled patch points.

Table 17: Static Patch Space Assembler Code

Platform	Assembler Code
Compaq Tru64 UNIX	<pre>.data .align 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .byte 0x00 : PATCH_SIZE TVDB_patch_end_address:</pre>

Table 17: Static Patch Space Assembler Code (cont.)

Platform	Assembler Code
IBM AIX	<pre>.csect .data{RW}, 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .space PATCH_SIZE TVDB_patch_end_address:</pre>
SGI IRIX	<pre>.data .align 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .space PATCH_SIZE TVDB_patch_end_address:</pre>

Here is how you would use the static patch space assembler code:

- 1 Use an ASCII editor and place the assembler code into a file named **tvdb\_patch\_space.s**.
- 2 Replace the **PATCH\_SPACE** tag with the decimal number of bytes you want. This value must be a multiple of 8.
- 3 Assemble the file into an object file by using a command such as:  

```
cc -c tvdb_patch_space.s
```

On SGI IRIX, use **-n32** or **-64** to create the correct object file type.
- 4 Link the resulting **tvdb\_patch\_space.o** into your program.

## Using Watchpoints

TotalView lets you monitor the changes that occur to memory locations by creating a special kind of action point called a *data watchpoint*, or just *watchpoint* for short. Watchpoints are most often used to find a statement in your program that is writing to a "stray" memory location. This can occur, for example, when processes share memory and more than one process writes to the same location. It can also occur when your program writes off the end of an array or when your program has a dangling pointer.



TotalView watchpoints are called *modify watchpoints* because TotalView only *triggers* a watchpoint when your program modifies a memory location. If a program writes a value into a location that is the same as what is already stored, TotalView does not trigger the watchpoint because the location's value did not change.

For example, if location 0x10000 has a value of zero and your program writes a zero into this location, TotalView does not trigger the watchpoint even though your program wrote data into the memory location. See "Triggering Watchpoints" on page 228 for more details on when watchpoints trigger.

TotalView also lets you create *conditional watchpoints*. A conditional watchpoint is similar to an evaluation point in that TotalView will evaluate an expression when the watchpoint triggers. You can use conditional watchpoints for a number of purposes. For example, you can use it to test if a value changes its sign—that is, it becomes positive or negative—or if a value moves above or below some threshold value.

## Architectures

The number of watchpoints, their size, and alignment restrictions differ from platform to platform. (This is because TotalView relies on the operating system and its hardware to implement data watchpoints.)

**NOTE** Watchpoints are not available on Alpha Linux and HP.

The following list describes constraints that exist on each platform:

**Compaq Tru64** Tru64 places no limitations on the number of watchpoints that you can create, and there are no alignment or size constraints. However, watchpoints cannot overlap, and you cannot create a watchpoint on an already write-protected page.

Watchpoints use a page protection scheme. Because the page size is 8,192 bytes, watchpoints can degrade performance if your program frequently writes to pages containing watchpoints.

<b>IBM AIX</b>	You can create one watchpoint on AIX 4.3.3.0-2 (AIX 4.3R) or later systems. (AIX 4.3R is available as APAR IY06844.) This watchpoint cannot be longer than 8 bytes and it must be aligned within an 8-byte boundary.
<b>IRIX6 MIPS</b>	Watchpoints are implemented on IRIX 6.2 and later operating systems. These systems allow you to create about 100 watchpoints. There are no alignment or size constraints. However, watchpoints cannot overlap.
<b>Linux x86</b>	You can create up to four watchpoints and each must be 1, 2, or 4 bytes in length, and a memory address must be aligned for the byte length. That is, a 4-byte watchpoint must be aligned on a 4-byte address boundary, and a 2-byte watchpoint must be aligned on a 2-byte boundary, etc.
<b>Solaris SPARC</b>	Watchpoints are implemented on Solaris 2.6 or later operating systems. These operating system allow you to create hundreds of watchpoints, and there are no alignment or size constraints. However, watchpoints cannot overlap.

Typically, a debugging session does not use many watchpoints. In most cases, only one memory location at a time is being monitored. So, restrictions on the number of values you can watch are seldom an issue.

## Creating Watchpoints

Watchpoints are created by selecting **Tools > Watchpoint** from within a Variable Window (If your platform does not support data watchpoints, this menu item is dimmed.)

After selecting this command, TotalView displays the dialog box shown in Figure 117.

Using this dialog box, you can create an unconditional or conditional watchpoint. The watchpoint you will set is set for the entire variable, not just for an element of it. For example, when you invoke this dialog box, the **Address** field is set to the variable's location in memory. If you want to look

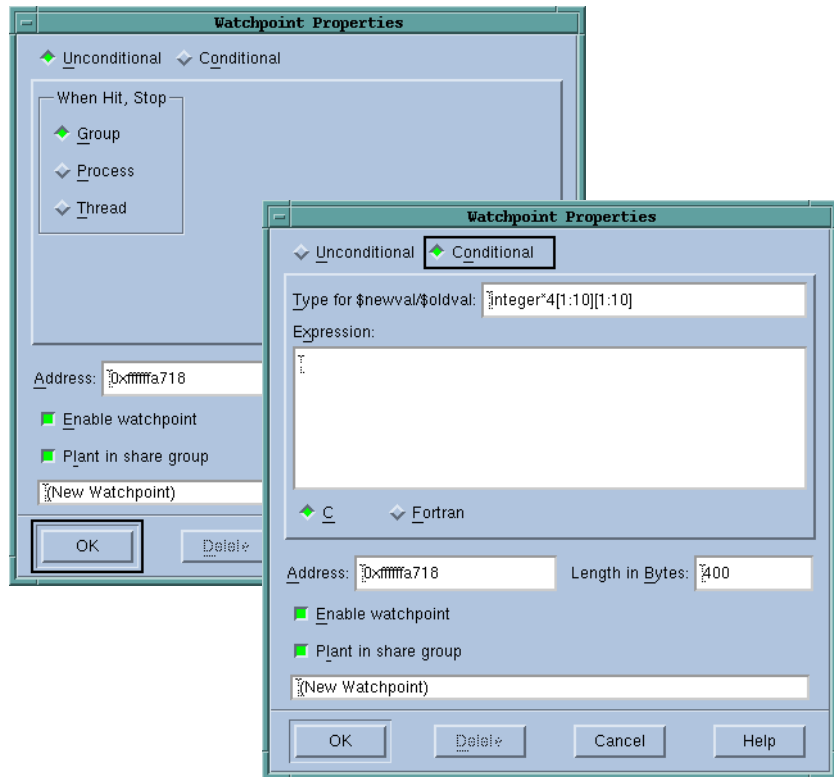


Figure 117: Tools > Watchpoint Dialog Boxes

at only one element of a structure or an array, simply dive on the element in the Variable Window before setting the watchpoint.

For information on the fields in this dialog box, see the online help.

## Displaying Watchpoints

The watchpoint entry, indicated by UDWP for Unconditional Data Watchpoint and CDWP for Conditional Data Watchpoint, displays the action point ID, the amount of memory being watched, and the location being watched.

If you dive into a watchpoint, TotalView displays the **Watchpoint Properties** dialog box.

If you select a watchpoint, TotalView will toggle the enabled/disabled state of the watchpoint.

## Watching Memory

A watchpoint tracks a memory location: it does not track a variable. This means that a watchpoint may not perform as you would expect it to when watching stack or automatic variables. For example, assume that you create a watchpoint to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is deallocated. At this time, TotalView is watching unallocated stack memory. And, when the stack memory is reallocated to a new stack frame, TotalView is still watching this same position. This means that TotalView triggers the watchpoint when something changes this newly allocated memory.

Also, if your program reinvokes a subroutine, it usually executes at a different part of the stack. So, if the subroutine changes, this change may not be seen because the variable is at a different memory location.

All of this means that in most circumstances, you cannot place a watchpoint on a stack variable. If you need to watch a stack variable, you will need to create and delete the watchpoint each time your program invokes the subroutine.

**NOTE** In some circumstances, a subroutine will always be called from the same location. This means that its local variables will probably be in the same location, so it may be worth trying.

If you place a watchpoint on a global or static variable that is always invoked by reference (that is, the value of a variable is always accessed using a pointer to the variable), you can set a watchpoint on it because the memory locations used by the variable are not changing.

## Triggering Watchpoints

When a watchpoint triggers, the thread's program counter (PC) points to the instruction *following* the instruction that caused the watchpoint to trigger. If the memory store instruction is the last instruction in a source statement, the PC will be pointing to the source line *following* the statement that triggered the watchpoint. (Breakpoints and watchpoints work differently. A

breakpoint stops *before* an instruction executes. In contrast, a watchpoint stops *after* an instruction executes.)

## Using Multiple Watchpoints

If a program modifies more than one byte with one program instruction or statement (which is normally the case when storing a word), TotalView triggers the watchpoint with the lowest memory location in the modified region. Although the program may be modifying locations monitored by other watchpoints, only the watchpoint for the lowest memory location is triggered. This can occur when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these locations.

For example, assume that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also assume that your program uses a single instruction to store a 2-byte value at locations 0x10000 and 0x10001. If the 2-byte storage operation modifies both bytes, the watchpoint for location 0x10000 triggers. The watchpoint for location 0x10001 does not and will not trigger at this time.

Here's a second example. Assume that you have a 4-byte integer that uses storage locations 0x10000 through 0x10003 and you set a watchpoint on this integer. If a process modifies location 0x10002, TotalView triggers the watchpoint. Now assume that you are watching two adjacent 4-byte integers that are stored in locations 0x10000 through 0x10007. If a process writes to locations 0x10003 and 0x10004 (that is, one byte within each), TotalView triggers the watchpoint associated with location 0x10003. The watchpoint associated with location 0x10004 does not trigger.

## Data Copies

TotalView keeps an internal copy of data in the watched memory locations for each process sharing the watchpoint. If you create watchpoints that cover a large area of memory or if your program has a large number of processes, you will increase TotalView's virtual memory requirements. Further, TotalView refetches data for each memory location whenever it continues the process or thread. This can affect TotalView's performance.

## Conditional Watchpoints

If you associate an expression with a watchpoint (by selecting the **CDWP** icon in the **Tools > Watchpoint** dialog box and entering an expression), TotalView will evaluate the expression after the watchpoint triggers. The programming statements that you can use in this area are identical to those that you can use when creating an evaluation point, except that you cannot call functions from a watchpoint expression.

The variables used in watchpoint expressions must be global. This is because the watchpoint can be triggered from any procedure or scope within your program.

Because memory locations are not scoped, the variable used in your expression must be globally accessible.

**NOTE** Fortran does not have global variables. Consequently, you cannot directly refer to your program's variables.

TotalView has two intrinsic variables that are specifically designed to be used with conditional watchpoint expressions:

<b>\$oldval</b>	The value of the memory locations before a change is made.
<b>\$newval</b>	The value of the memory locations after a change is made.

Here is an expression that uses these values:

```
if (iValue != 42 && iValue != 44) {
    iNewValue = $newval; iOldValue = $oldval; $stop;}
```

When the value **iValue** global variable is neither 42 nor 44, TotalView will store the new and old memory values in the **iNewValue** and **iOldValue** variables. These variables are defined in the program. (Storing the old and new values is a convenient way of letting you monitor the changes made by your program.)

Here is a condition that triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And here's a condition that triggers a watchpoint when the sign of the value in the memory location changes:

```
if ($newval * $oldval <= 0) $stop
```

Both of these examples require that you set the **Type for \$oldval/\$newval** field in the **Watchpoint Properties** dialog box.

For more information on writing expressions, see *"Writing Code Fragments"* on page 234.

If a watchpoint has the same length as the **\$oldval** or **\$newval** data type, the value of these variables is apparent. However, if the data type is shorter than the length of the watch region, TotalView searches for the first changed location in the watched region and uses that location for **\$oldval** and **\$newval** variables. (It aligns data within the watched region based on the size of the data's type. For example, if the data type is a 4-byte integer and byte 7 in the watched region changes, TotalView uses bytes 4 through 7 of the watchpoint when it assigns values to these variables.)

For example, suppose you are watching an array of 1000 integers called **must\_be\_positive** and you want to trigger a watchpoint as soon as one element becomes negative. You would declare the type for **\$oldval** and **\$newval** to be **int** and use the following condition:

```
if ($newval < 0) $stop;
```

When your program writes a new value to the array, TotalView triggers the watchpoint, sets the values of **\$oldval** and **\$newval**, and evaluates the expression. When **\$newval** is negative, the **\$stop** statement halts the process.

This can be a very powerful technique for range checking all the values written into an array. (Because of byte length restrictions, you can only use this technique on IRIX and Solaris.)

**NOTE** Conditional watchpoints are always interpreted by TotalView; they are never compiled. And, because interpreted watchpoints are single threaded within TotalView, every process or thread that writes to the watched location must wait for other instances of the watchpoint to finish executing. This can adversely affect performance.

## Saving Action Points to a File

You can save a program's action points into a file. TotalView will then use this information to reset these points when you restart the program. When you save action points, TotalView creates a file named *program.TVD.breakpoints*, where *program* is the name of your program.

**NOTE** Watchpoints are not saved.

Use the **Action Point > Save All** command to save your action points to a file. TotalView places the action points file in the same directory as your program.

If you are using a preference to automatically save breakpoints, TotalView will automatically save action points to a file. Alternatively, starting TotalView with the **-sb** option (see "TotalView Command Syntax" on page 289) also tells TotalView to save your breakpoints.

Automatic saving and loading is controlled by preferences (see **File > Preferences** within the online help for more information) and can be altered temporarily by using the **-sb** and **-nlb** options.

## Evaluating Expressions

TotalView lets you open a window for evaluating expressions in the context of a particular process and evaluate expressions in C, Fortran, or assembler.

**NOTE** Not all platforms let you use assembler constructs; see Appendix C "Architectures" on page 337 for details.

You can use the **Tools > Evaluate** dialog box in many different ways, but here are two examples:

- Expressions can contain loops, so you could use a **for** loop to search an array of structures for an element set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression field.



- Because you can call subroutines, you can test and debug a single routine in your program without building a test program to call it.

To evaluate an expression:

- 1 Tell TotalView to display the **Evaluate** dialog box by selecting the **Tools > Evaluate** command. An Evaluate dialog box appears. If your program has not yet been created, you will not be able to use any of the program's variables or call any of its functions.
- 2 Select the button (if it is not already selected) for the language in which you will write the code.
- 3 Move to the **Expression** field and enter a code fragment. For a description of the supported language constructs, see "Writing Code Fragments" on page 234.

TotalView returns the value of the last statement in the code fragment. This means that you do not have to assign the expression's return value to a variable. Figure 118 shows a sample expression. The last statement in this example assigns the value of **my\_var1-3** back to **my\_var1**. Because this is the last statement in the code fragment, the value placed in the Result field would be the same if you had just typed **my\_var1-3**.

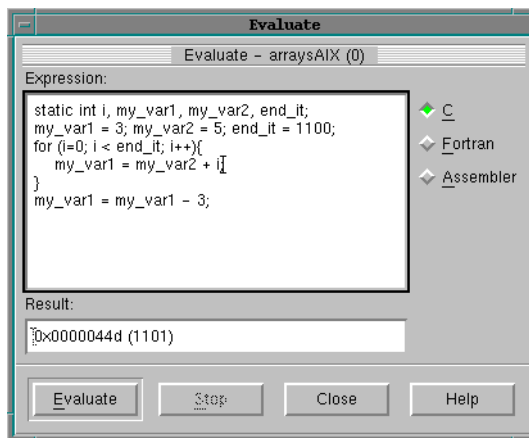


Figure 118: **Tools > Evaluate** Dialog Box

- 4 Select the **Evaluate** button. If TotalView finds an error, it places the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the **Result** field.

While the code is being executed, you cannot modify anything in the dialog box. TotalView may also display a message box that tells you that it is waiting for the command to complete. (See Figure 119.)

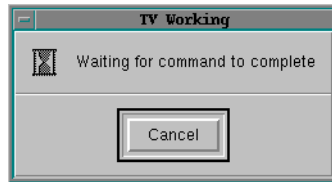


Figure 119: **Waiting to Complete Message Box**

If you select **Cancel**, TotalView stops execution.

Since TotalView evaluates code fragments in the context of the target process, it evaluates stack variables according to the currently selected stack frame. If the fragment reaches a breakpoint (or stops for any other reason), TotalView stops evaluating your expression. Assignment statements within an expression can affect the target process because they can change a variable's value.

## Writing Code Fragments

You can use code fragments in evaluation points and in the **Tools > Evaluate** dialog box. This section describes the intrinsic variables, built-in statements, and language constructs supported by TotalView.

### Intrinsic Variables

The TotalView expression system supports built-in variables that allow you to access special thread and process values. All variables are 32-bit integers, which is an **int** or a **long** on most platforms. Table 18 lists the intrinsic variable names and their meanings.

Table 18: **Intrinsic Variables**

Name	Returns
\$clid	The cluster ID. (Interpreted expressions only.)
\$duid	The TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.)

Table 18: Intrinsic Variables (cont.)

Name	Returns
<b>\$newval</b>	The value just assigned to a watched memory location. (Watchpoints only.)
<b>\$nid</b>	The node ID. (Interpreted expressions only.)
<b>\$oldval</b>	The value that existed in a watched memory location before a new value modified it. (Watchpoints only.)
<b>\$pid</b>	The process ID.
<b>\$processduid</b>	The DUID of the process. (Interpreted expressions only.)
<b>\$systid</b>	The system-assigned thread ID. When this is referenced from a process, TotalView throws an error.
<b>\$tid</b>	The TotalView-assigned thread ID. When this is referenced from a process, TotalView throws an error.

Intrinsic variables allow you to create thread-specific breakpoints from the expression system. For example, the **\$tid** intrinsic variable and the **\$stop** built-in operation let you create thread-specific breakpoint as follows:

```
if ($tid == 3)
    $stop;
```

This tells TotalView to stop the process only when thread 3 evaluates the expression.

You can also create complex expressions by using intrinsic variables. For example:

```
if ($pid != 34 && $tid > 7)
    printf ("Hello from %d.%d\n", $pid, $tid);
```

**NOTE** Using any of the following intrinsics means that the evaluation point is interpreted instead of compiled: **\$clid**, **\$duid**, **\$nid**, **\$processduid**, **\$systid**, **\$tid**, and **\$visualize**. In addition, **\$pid** forces interpretation on AIX.

You cannot assign a value to an intrinsic variable or obtain their address.

## Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are shown in the following table.

Table 19: Built-In Statements Used in Expressions

Statement	Use
<b>\$count</b> <i>expression</i>	Sets a process-level countdown breakpoint.
<b>\$countprocess</b> <i>expression</i>	When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the process stops. The other processes in the control group continue to execute.
<b>\$countall</b> <i>expression</i>	Sets a program-group-level countdown breakpoint. All processes in the control group stop when any process in the group executes this statement for the number of times specified by <i>expression</i> .
<b>\$countthread</b> <i>expression</i>	<p>Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by <i>expression</i>, the thread stops. Other threads in the process continue to execute.</p> <p>If the target system can not stop an individual thread, this statement performs identically to <b>\$countprocess</b>.</p> <p>A thread evaluates <i>expression</i> when it executes <b>\$count</b> for the first time. This expression must evaluate to a positive integer. When TotalView first encounters this intrinsic, it determines a value for <i>expression</i>. TotalView will not reevaluate until the expression actually stops the thread. This means that TotalView ignores changes in the value of <i>expression</i> until it hits the breakpoint. After the breakpoint occurs, TotalView reevaluates <i>expression</i> and sets a new value for this statement.</p>

Table 19: Built-In Statements Used in Expressions (cont.)

Statement	Use
	The internal counter is stored in the process and shared by all threads in that process.
<b>\$hold</b> <b>\$holdprocess</b>	Holds the current process. If all other processes in the group are already held at this <b>Eval</b> point, then TotalView will release all of them. If other processes in the group are running, they continue to run.
<b>\$holdstopall</b> <b>\$holdprocessstopall</b>	Exactly like <b>\$hold</b> , except any processes in the group which are running are <i>stopped</i> . Note that the other processes in the group are not automatically held by this call—they are just stopped.
<b>\$holdthread</b>	Freezes the current thread, leaving other threads running.
<b>\$holdthreadstop</b> <b>\$holdthreadstopprocess</b>	Exactly like <b>\$holdthread</b> except it <i>stops</i> the <i>process</i> . The other processes in the group are left running.
<b>\$holdthreadstopall</b>	Exactly like <b>\$holdthreadstop</b> except it stops the entire group.
<b>\$stop</b> <b>\$stopprocess</b>	Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.
<b>\$stopall</b>	Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.
<b>\$stopthread</b>	Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute.
	If the target system can not stop an individual thread, this statement performs identically to <b>\$stopprocess</b> .

Table 19: Built-In Statements Used in Expressions (cont.)

Statement	Use
<code>\$visualize(expression[,slice])</code>	Visualizes the data specified by <i>expression</i> and modified by the optional <i>slice</i> value. <i>Expression</i> and <i>slice</i> must be expressed using the code fragment's language. The expression must return a dataset (after modification by <i>slice</i> ) that can be visualized. <i>slice</i> is a quoted string containing a slice expression. For more information on using <code>\$visualize</code> in an expression, see "Visualizing Data in Expressions" on page 252.

## C Constructs Supported

When writing code fragments in C, keep these guidelines in mind:

- C-style (`/* comment */`) and C++-style (`// comment`) comments are permitted. For example:

```
// This code fragment creates a temporary patch
i = i + 2; /* Add two to i */
```

- You can omit semicolons if the result is not ambiguous.
- You can use dollar signs (\$) in identifiers.

## Data Types and Declarations

The following list describes the C data types and declarations that you can use:

- The data types that you can use are **char**, **short**, **int**, **float**, **double**, and pointers to any primitive type or any named type in the target program.
- Only simple declarations are permitted. Do not use **struct**, **union**, and array declarations.
- You can refer to variables of any type in the target program.
- Unmodified variable declarations are considered local. References to these declarations override references to similarly named global variables and other variables in the target program.
- (Compiled evaluation points only.) The **global** declaration makes a variable available to other evaluation points and expression windows in the target process.

- (Compiled evaluation points only.) The **extern** declaration references a global variable that was or will be defined elsewhere. If the global variable is not yet defined, TotalView displays a warning.
- Static variables are local and persist even after TotalView evaluates an evaluation point.
- TotalView only evaluates expressions that initialize static and global variables the first time it evaluates a code fragment. In contrast, it initializes local variables each time it evaluates a code fragment.

## Statements

The following list describes the C language statements that you can use.

- The statements that you can use are assignment, **break**, **continue**, **if/else** structures, **for**, **goto**, and **while**.
- You can use the **goto** statement to define and branch to symbolic labels. These labels are local to the window. You can also refer to a line number in the program. This line number is the *tag field* number of the source code line. For example, here is a **goto** statement that branches to source line number 432 of the target program:

```
goto 432;
```

- Although function calls are permitted, you cannot pass structures.
- Type casting is permitted.

All operators are permitted, with these limitations:

- TotalView does not support the **?:** conditional operator.
- While you can use the **sizeof** operator, you cannot use it for data types.
- The **(type)** operator cannot cast data to fixed-dimension array by using C cast syntax.

## Fortran Constructs Supported

When writing code fragments in Fortran, keep these guidelines in mind:

- Only enter one statement on a line. You cannot continue a statement onto more than one line.
- You can use **GOTO**, **GO TO**, **ENDIF**, and **END IF** statements; while **ELSEIF** is not allowed, you can use **ELSE IF**.

- Syntax is free-form. No column rules apply.
- You can enter comments in several formats. For example, you can use the following format:

```
C I=I+1
/*
I=I+1
J=J+1
ARRAY1(I,J)= I * J
*/
```

- The space character is significant and is sometimes required. (Some Fortran 77 compilers ignore all space characters.) For example:

Valid	Invalid
DO 100 I=1,10	DO100I=1,10
CALL RINGBELL	CALL RING BELL
X.EQ. 1	X.EQ.1

## Data Types and Declarations

The following is a list of data types and declarations that you can use in a Fortran expression.

- You can use the following data types: **INTEGER** (assumed to be **long**), **REAL**, **DOUBLE PRECISION**, and **COMPLEX**.
- You cannot use implied data types are not permitted.
- You can only use simple declarations. You cannot use a **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, or an array declaration.
- You can refer to variables of any type in the target program.

## Statements

The following list describes the Fortran language statements that you can use.

- You can use the following statements: assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not an alternate return).



- A **GOTO** statement can refer to a line number in your program. This line number is the *tag field* number. For example, the following **GOTO** statement branches to source line number 432:

**GOTO \$432;**

You must use a dollar sign before the line number so that TotalView knows that you are referring to the tag field number rather than a statement label.

- The only expression operators that are not supported are the **CHARACTER** operators and the **.EQV.**, **.NEQV.**, and **.XOR.** logical operators.
- You cannot use subroutine function and entry definitions.
- You cannot use Fortran 90 array syntax.
- You cannot use Fortran 90 pointer assignment (the **=>** operator).
- You cannot call Fortran 90 functions that require assumed shape array arguments.

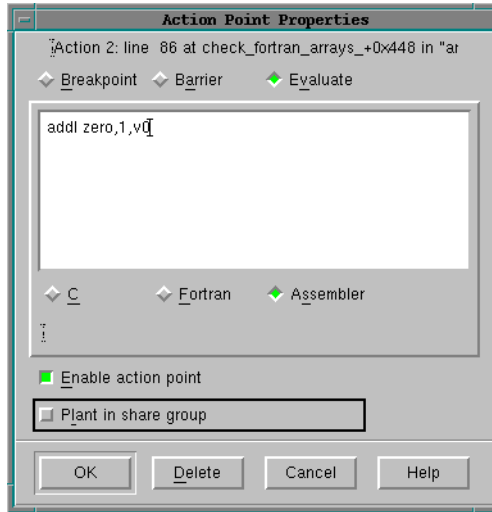
## Writing Assembler Code

On Compaq Tru64 UNIX, RS/6000 IBM AIX, and SGI IRIX operating systems, TotalView lets you use assembler code in evaluation points, conditional breakpoints, and in the **Tools > Evaluate** dialog box. However, if you want to use assembler constructs, you must enable compiled expressions. See “*Interpreted vs. Compiled Expressions*” on page 220 for instructions.

To indicate that an expression in the breakpoint or Evaluate dialog box is an assembler expression, click on the **Assembler** button in the **Action Point > Properties** dialog box, as shown in Figure 120.

Assembler expressions are written in the TotalView Assembler Language. In this language, instructions are written in the target machine’s native assembler language; the operators available to construct expressions in instruction operands and the set of available pseudo-operators, however, are the same on all machines.

The TotalView assembler accepts instructions using the same mnemonics recognized by the native assembler and recognizes the same names for registers that native assemblers recognize.



**Figure 120: Using Assembler**

Some architectures provide extended mnemonics that do not correspond exactly with machine instructions and which represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView assembler recognizes these mnemonics if:

- They assemble to exactly one instruction.
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence.

Assembler Language, labels are indicated as *name:* and appear at the beginning of a line. Labels may appear alone on a line. The symbols you can use include labels defined in the assembler expression and all program symbols.

The TotalView assembler operators are described in the following table:

Table 20: TotalView Assembler Operators

Operators	Definition
+	Plus
–	Minus (also unary)
*	Times
#	Remainder
/	Quotient
&	Bitwise AND
^	Bitwise XOR
!	Bitwise OR NOT (also unary -, bitwise NOT)
	Bitwise OR
( <i>expr</i> )	Grouping
<<	Left shift
>>	Right shift
" <i>text</i> "	Text string, 1-4 characters long, is right justified in a 32-bit word
hi16 ( <i>expr</i> )	Low 16 bits of operand <i>expr</i>
hi32 ( <i>expr</i> )	High 32 bits of operand <i>expr</i>
lo16 ( <i>expr</i> )	High 16 bits of operand <i>expr</i>
lo32 ( <i>expr</i> )	Low 32 bits of operand <i>expr</i>

The TotalView Assembler pseudo-operations are as follows:

Table 21: TotalView Assembler Pseudo Operations

Pseudo Ops	Definition
\$debug   0   1	<i>Internal debugging option.</i> With no operand, toggle debugging; 0 => turn debugging off 1 => turn debugging on
\$hold	Hold the process
\$holdprocess	
\$holdstopall	Hold the process and stop the control group
\$holdprocessstopall	

Table 21: TotalView Assembler Pseudo Operations (cont.)

Pseudo Ops	Definition
<b>\$holdthread</b>	Hold the thread
<b>\$holdthreadstop</b>	Hold the thread and stop process
<b>\$holdthreadstopprocess</b>	
<b>\$holdthreadstopall</b>	Hold the thread and stop the control group
<b>\$long_branch</b> <i>expr</i>	Branch to location <i>expr</i> using a single instruction in an architecture-independent way; using registers is not required
<b>\$stop</b>	Stop the process
<b>\$stopprocess</b>	
<b>\$stopall</b>	Stop the control group
<b>\$stopthread</b>	Stop the thread
<i>name</i> = <i>expr</i>	Same as <b>def</b> <i>name</i> , <i>expr</i>
<b>align</b> <i>expr</i> [, <i>expr</i> ]	Align location counter to an operand 1 alignment; use operand 2 (or zero) as the fill value for skipped bytes
<b>ascii</b> <i>string</i>	Same as <i>string</i>
<b>asciz</b> <i>string</i>	Zero-terminated string
<b>bss</b> <i>name</i> , <i>size-expr</i> [, <i>expr</i> ]	Define <i>name</i> to represent <i>size-expr</i> bytes of storage in the <b>bss</b> section with alignment optional <i>expr</i> ; the default alignment depends on the size: if <i>size-expr</i> >= 8 then 8 else if <i>size-expr</i> >= 4 then 4 else if <i>size-expr</i> >= 2 then 2 else 1
<b>byte</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of bytes
<b>comm</b> <i>name</i> , <i>expr</i>	Define <i>name</i> to represent <i>expr</i> bytes of storage in the <b>bss</b> section; <i>name</i> is declared global; alignment is as in <b>bss</b> without an alignment argument
<b>data</b>	Assemble code into data section (data)
<b>def</b> <i>name</i> , <i>expr</i>	Define a symbol with <i>expr</i> as its value
<b>double</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of doubles
<b>equiv</b> <i>name</i> , <i>name</i>	Make operand 1 be an abbreviation for operand 2

Table 21: TotalView Assembler Pseudo Operations (cont.)

Pseudo Ops	Definition
<b>fill</b> <i>expr</i> , <i>expr</i> , <i>expr</i>	Fill storage with operand 1 objects of size operand 2, filled with value operand 3
<b>float</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of floats
<b>global</b> <i>name</i>	Declare <i>name</i> as global
<b>half</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of 16-bit words
<b>lcomm</b> <i>name</i> , <i>expr</i> [, <i>expr</i> ]	Identical to <b>bss</b>
<b>lsym</b> <i>name</i> , <i>expr</i>	Same as <b>def</b> <i>name</i> , <i>expr</i> but allows redefinition of a previously defined name
<b>org</b> <i>expr</i> [, <i>expr</i> ]	Set location counter to operand 1 and set operand 2 (or zero) to fill skipped bytes
<b>quad</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of 64-bit words
<b>string</b> <i>string</i>	Place <i>string</i> into storage
<b>text</b>	Assemble code into text section (code)
<b>word</b> <i>expr</i> [, <i>expr</i> ] ...	Place <i>expr</i> values into a series of 32-bit words
<b>zero</b> <i>expr</i>	Fill <i>expr</i> bytes with zeros



# Visualizing Data



The TotalView Visualizer works with the TotalView debugger to create graphical images of your program's array data. Topics in this chapter are:

- How the Visualizer Works
- Configuring TotalView to Launch the Visualizer
- Data Types That TotalView Can Visualize
- Visualizing Data from the Variable Window
- Visualizing Data in Expressions
- Using the TotalView Visualizer
- Viewing of Data
- Launching the Visualizer from the Command Line

The Visualizer is not available on Linux Alpha and 32-bit SGI Irix.

**NOTE** The online help contains information on adapting a third party visualizer so that it can be used with TotalView.

## How the Visualizer Works



You can use the Visualizer in two ways: you can launch it from within TotalView to visualize data as you debug your programs, and you can run it from a command line to visualize data previously dumped to a file.

Visualizing your program's data is a two step process:

- 1 You interact with TotalView to choose the data being visualized.
- 2 You interact with the Visualizer to choose how it should display your data.

The TotalView debugger handles the first of these interactions, extracting data and marshalling it into a standard format that it sends down a pipe. The Visualizer then reads the data from this pipe and displays it for analysis. The following figure shows this relationship.

TotalView: Extracts data from an array

The TotalView Visualizer: Displays the array data graphically

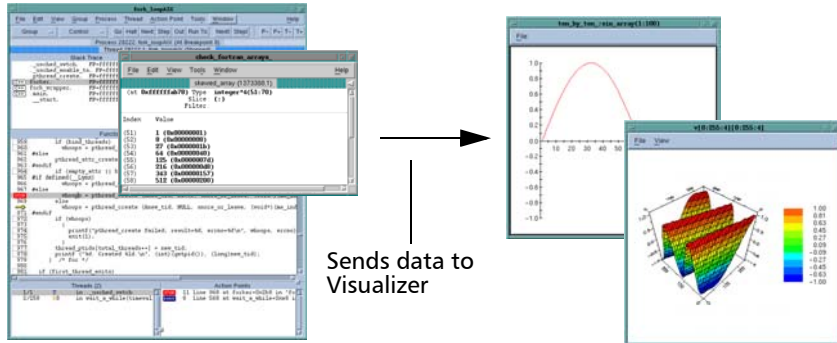


FIGURE 121: TotalView Visualizer Connection

You can send data directly from TotalView to the Visualizer while you are debugging your program or you can send data from TotalView directly to a third-party visualizer. If you save visualization data to a file, you can launch the Visualizer from the command line to have it visualize this saved data. Figure 122 shows these relationships.

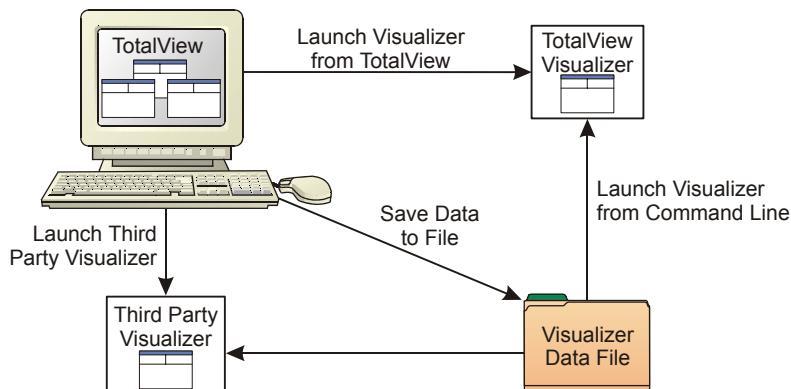


FIGURE 122: TotalView Visualizer Relationships



## Configuring TotalView to Launch the Visualizer

TotalView can automatically launch the Visualizer when it is named in a Variable window, or the **Action Point > Properties** or the **Tools > Evaluate** dialog boxes. After TotalView launches the Visualizer, it sends data to the Visualizer's so you can visualize datasets as your program creates them.

If you disable visualization, TotalView silently ignores all attempts to use the Visualizer. This is useful when you want to execute some code containing evaluation points that do visualization and do not want to individually disable all these points.

To change the Visualizer launch options interactively, select **File > Preferences**, then select the **Launch Strings** Tab. This is shown in Figure 123.

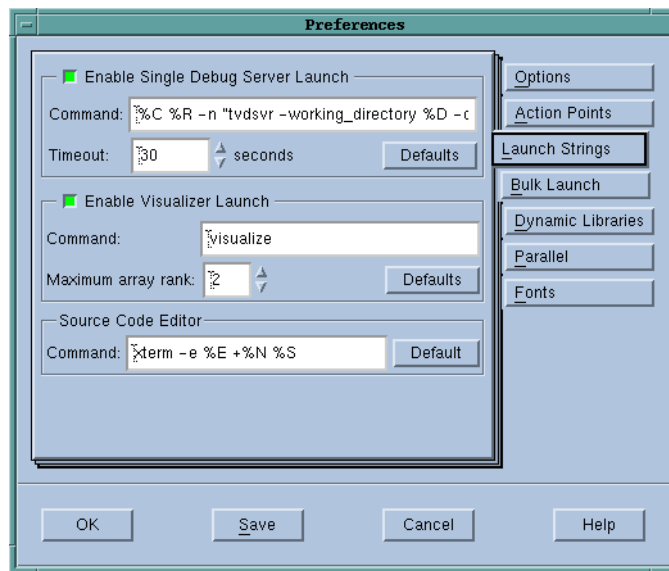


FIGURE 123: **File > Preferences Launch Strings Page**

You can now:

- If you use a customized command to start a visualizer, enter its startup command in the **Command** edit box. Entering information in this field is discussed a little later in this section.

- Change the auto launch option. If you want to disable visualization, clear the **Enable Visualizer Launch** check box.
- Change the maximum permissible rank. Edit the value in the **Maximum array rank** edit field to save the data exported from the debugger or display it in a different visualizer. You can enter a number from **1** to **16**.  
Setting the maximum permissible rank to either 1 or 2 (the default is 2) ensures that the TotalView Visualizer can use your data—the Visualizer displays only two dimensions of data. This limit does not apply to data saved in files or to visualizers that can display more than two dimensions of data.
- Clicking on the **Defaults** button sets options to their defaults. This reverts options to their defaults even if you have used X resources to change them.

If you disable visualization while the Visualizer is running, TotalView closes its connection to the Visualizer. If you reenables visualization, TotalView launches a new Visualizer process the next time you visualize something.

You can change the shell command that TotalView uses to launch the visualizer by editing the Visualizer launch command. (You would use this if you are running a different visualizer.) Or, you can save this information for viewing at another time. For example, you can save visualization information by entering the following command:

```
cat > your_file
```

Later, you can visualize this information by using either of the following commands:

```
visualize -persist < your_file
visualize -file your_file
```

You can preset the visualizer launch options by setting X resources. For details, see Chapter 12 “X Resources” on page 275.

## Data Types That TotalView Can Visualize

The data selected for visualization is called a *dataset*. Each dataset is tagged with a TotalView-generated numeric identifier that lets the Visualizer know whether it is seeing a new dataset or an update to an existing dataset.

TotalView treats stack variables at different recursion levels or call paths as different datasets

TotalView can visualize one- and two-dimensional arrays of character, integer, or floating-point data. This data cannot be located in registers. If an array has more than two dimensions, you can visualize part of it using an array slice expression that creates a sub-array having fewer dimensions. Figure 124 shows a three-dimensional variable sliced into two dimensions by selecting a single index in the middle dimension.

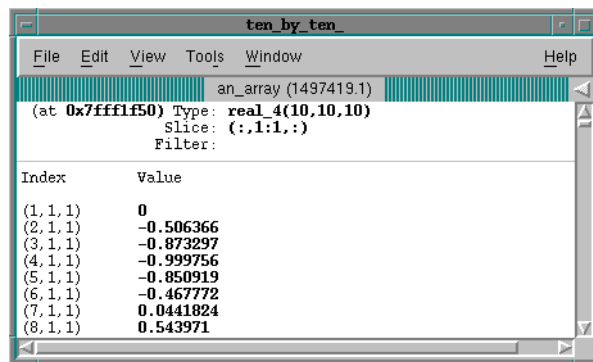


FIGURE 124: A Three-Dimensional Array Sliced into Two Dimensions

## Visualizing Data from the Variable Window

You can tell the Visualizer to display the information contained within a Variable Window. Before you can visualize an array, you must open a Variable Window for the array's data and stop program execution at the point where you want to visualize the array's values. Figure 125 is an example.

Editing the **Type** and **Slice** fields lets you select the data you want visualized. For example, editing the **Slice** fields limits the amount of data being visualized. (See "Displaying Array Slices" on page 183.) Limiting the amount increases the Visualizer's speed.

Launch the Visualizer by selecting the **Tools > Visualize** command. The Visualizer will then create its window. The data sent to the Visualizer is not

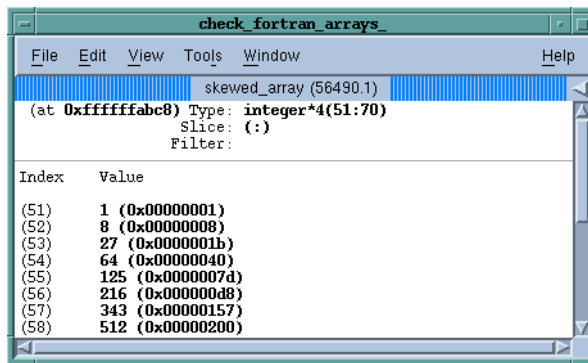


FIGURE 125: Variable Window

automatically updated as you step through your program. You must explicitly update the display by reentering the **Tools > Visualize** command.

You can also visualize a laminated variable. (See “*Visualizing a Laminated Variable Window*” on page 200.) The process or thread index forms one of the dimensions of the visualized data. This means that you can only visualize scalar or vector information. If you do not want the process or thread index as a dimension, use a nonlaminated display.

## Visualizing Data in Expressions

The **\$visualize** intrinsic (built-in) function lets you use TotalView’s evaluation system to visualize data. This function lets you:

- Visualize several different variables from a single expression.
- Visualize variables from the Process Window’s **Tools > Evaluate** dialog box.
- Visualize one or more variables from an evaluation point.

The syntax for the **\$visualize** intrinsic is:

```
$visualize ( array [, slice_string ] )
```

The *array* parameter is an expression naming the dataset being visualized. The optional *slice\_string* parameter is a quoted string defining a constant slice expression that modifies the *array* parameter’s dataset.

The following examples show how you can use this intrinsic. Notice that the array's dimension ordering differs in C and in Fortran.

<b>C</b>	<pre>\$visualize(my_array); \$visualize (my_array,"[::2][10:15]"); \$visualize (my_array,"[12][:1]");</pre>
<b>Fortran</b>	<pre>\$visualize (my_array) \$visualize (my_array,'(1 1:16,::2)') \$visualize (my_array,'(:,13)')</pre>

The first example in each programming language group visualizes the entire array. The second example selects every second element in the array's major dimension; it also clips the minor dimension to all elements in the given (inclusive) range. The third example reduces the dataset to a single dimension by selecting one subarray.

You may need a cast expression to let TotalView know what the dimensions of the variable being visualized are. For example, here is a procedure that passes a two-dimensional array parameter that does not specify the extent of the major dimension:

```
void my_procedure (double my_array[][32])
{ /* procedure body */ }
```

You will need to use the following cast expression because the first dimension is not specified:

```
$visualize (*(double[32][32]*)my_array);
```

You can use **\$visualize** in the **Tools > Evaluate** dialog box or by adding an expression to an evaluation point. But note that TotalView cannot compile an evaluation point or expression that contains **\$visualize**. Instead, TotalView interprets these statements. See *"Defining Evaluation Points"* on page 216 for information about compiled and interpreted expressions.

Using **\$visualize** in a **Tools > Evaluate** dialog box is a handy technique you can use to refine an array and slice arguments or to update a display of several arrays simultaneously.

## Visualizer Animation

Using **\$visualize** in an evaluation point lets you animate the changes that occur in your data because the Visualizer will update the array's display every time TotalView reaches the evaluation point.

## Using the TotalView Visualizer

The Visualizer has two types of windows:

### ■ Data Windows

The windows containing images of the datasets. By interacting with a Data Window, you can change its appearance and set viewing options.

### ■ A Directory Window

A window that lists the datasets that you can visualize. Use this window to set global options and to create views of your datasets. Using the Directory Window, you can open several Data Windows on a single dataset to get different views of the same data.

Figure 126 shows a Directory Window and two Data Windows. The left window shows a surface view while the right Data Window shows a graph view.

## Directory Window

The Directory Window contains a list of the datasets you can display. You can select a dataset by clicking on it and you can only select one dataset at a time. The **View** menu lets you select **Graph** or **Surface** visualization. Whenever TotalView sends a new dataset, the Visualizer updates its list of datasets. To delete a dataset from the list, click on it, and then display the **File** menu and select **Delete**.

You can automatically visualize a dataset by double-clicking on it.

The following list defines the Directory Window's menu bar commands.

- |                         |   |
|-------------------------|---|
| <b>File &gt; Delete</b> | Deletes the currently selected dataset. It removes the dataset from the dataset list and destroys any Data Windows displaying it. |
| <b>File &gt; Exit</b>   | Closes all windows and exits the <b>Visualizer</b> .  |

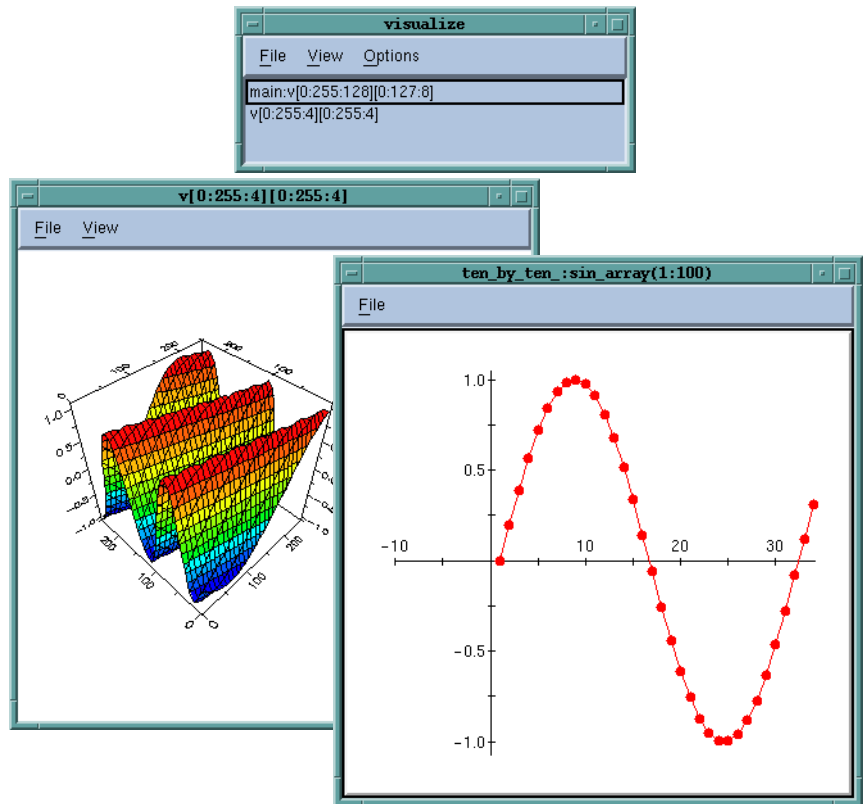


FIGURE 126: Sample Visualizer Windows

- View > Graph** Creates a new Graph Window; see "Graph Window" on page 258 for more detail.
- View > Surface** Creates a new Surface Window; see "Surface Window" on page 260 for more detail.
- Options > Auto Visualize**  
This item is a toggle; when enabled, the **Visualizer** automatically visualizes new datasets as they are read.

## Data Windows

Data Windows display graphical images of your data. Figure 127 shows a surface view and a graph view. Every Data Window contains a menu bar and a drawing area. The Data Window title is its dataset identification.

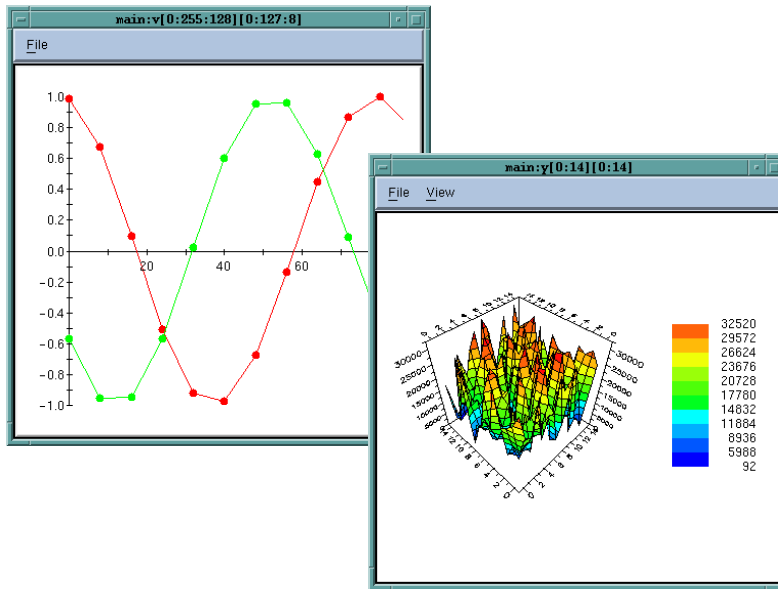


FIGURE 127: Sample Visualizer Data Windows

The **File** menu on the menu bar is the same for all Data Windows. Other items are unique to the type of Data Window. The Data Window menu commands are as follows.

- File > Close** Closes the Data Window.
- File > Delete** Deletes the Data Window's dataset from the dataset list. This also destroys other Data Windows viewing the dataset.
- File > Directory** Raises the Directory Window to the front of the desktop. If the Directory Window is minimized, the Visualizer restores it.
- File > New Base Window** Creates a new Data Window having the same visualization method and dataset as the current Data Window.
- File > Options** Pops up a window of viewing options.

The drawing area displays the image of your data. You can interact with the drawing area to alter the view of your data. For example, in the surface view, you can rotate the graph to view it from different angles. You can also



get the value and indices of the dataset element nearest the cursor by clicking on it. A pop-up window displays the information.

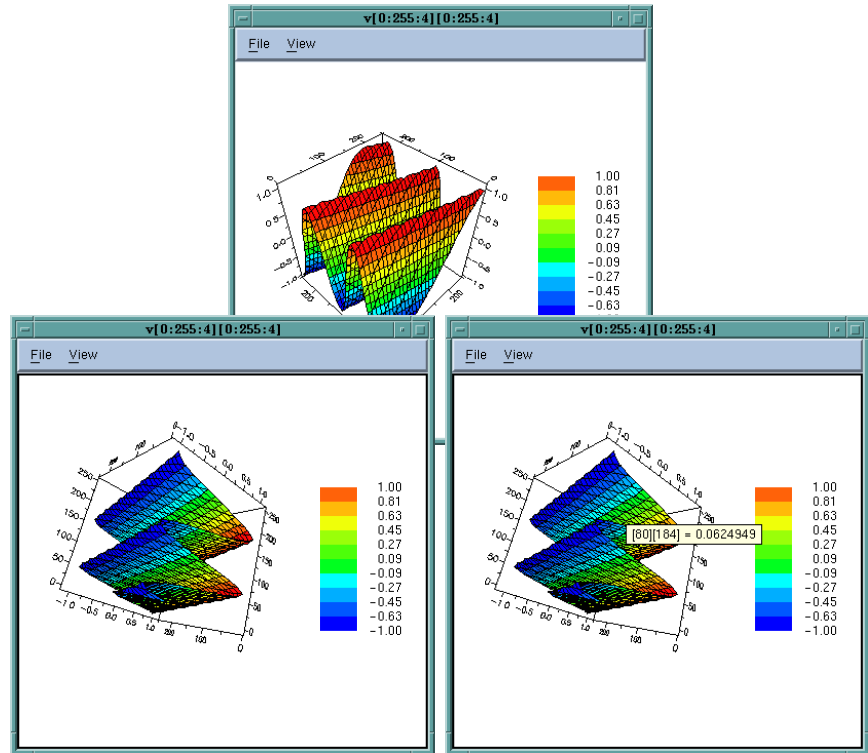


FIGURE 128: Rotating and Querying

## Viewing of Data

Different datasets can require different views to display their data. For example, a graph is more suitable for displaying one-dimensional datasets or two-dimensional datasets where one of the dimensions has a small extent; however, a surface view is better for displaying a two-dimensional dataset.

When the Visualizer is launched, one of the following actions will occur:

- If the dataset has never been visualized, the Visualizer chooses a method, based on how well a given dataset matches an ideal dataset for each method.

- If a dataset was previously visualized but no Data Window currently exists for it, the Visualizer creates a new Data Window by using the most recent visualization method.
- If a Data Window is currently displaying the dataset, the Visualizer raises it to the top of the desktop. If the window was minimized, the Visualizer restores it.

The Visualizer can automatically choose a visualization method and create a new Data Window when it reads a new dataset. You can enable and disable this feature from the **Options** menu in the Directory Window.

## Graph Window

The Graph Window displays a two-dimensional graph of one- or two-dimensional datasets. If the dataset is two-dimensional, the Visualizer displays multiple graphs. When you first create a Graph Window on a two-dimensional dataset, the Visualizer uses the dimension with the larger number of elements for the **x** axis. It then draws a separate graph for each subarray having the smaller number of elements. If you do not like this choice, you can transpose the data.

**NOTE** You probably do not want to use a graph to visualize two-dimensional datasets with large extents in both dimensions, as the display will be very cluttered.

You can display graphs with markers for each element of the dataset, with lines connecting dataset elements, or with both lines and markers as shown in Figure 129. See “*Displaying Graphs*” on page 258 for more details. Multiple graphs are displayed in different colors. The **X** axis of the graph is annotated with the indices of the long dimension. The **Y** axis shows you the data value.

You can scale and translate the graph, or pop up a window displaying the indices and values for individual dataset elements. See “*Manipulating Graphs*” on page 260 for details.

## Displaying Graphs

The **File > Options** dialog box lets you control how the Visualizer displays the graph. (See Figure 130.)

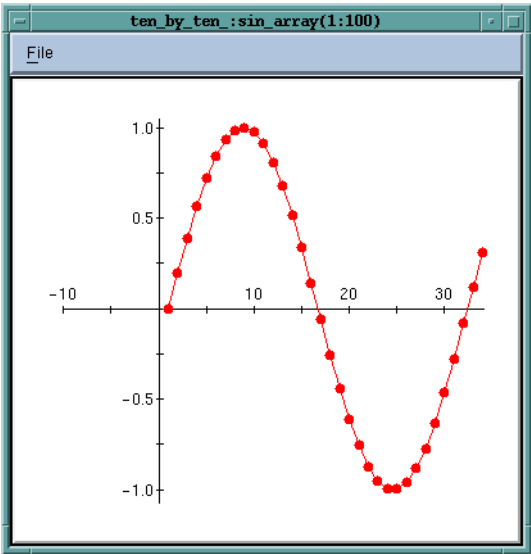


FIGURE 129: Visualizer Graph Data Window

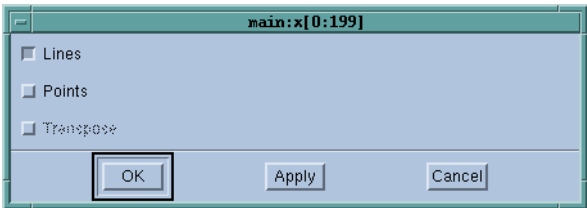


FIGURE 130: Graph Options Dialog Box

The following indicates the meanings of the check boxes within this dialog box.

- |                  |   |
|------------------|---|
| <b>Lines</b>     | If set, the Visualizer displays lines connecting dataset elements.                    |
| <b>Points</b>    | If set, the Visualizer displays markers for dataset elements.                         |
| <b>Transpose</b> | If set, the Visualizer inverts the <b>x</b> and <b>y</b> axis of the displayed graph. |

## Manipulating Graphs

You can manipulate the way the graph is displayed by using the following actions:

<b>Scale</b>	Press the <b>Control key</b> and hold down the <b>middle mouse</b> button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
<b>Translate</b>	Press the <b>Shift key</b> and hold down the <b>middle mouse</b> button. Moving the mouse drags the graph.
<b>Zoom</b>	Press the <b>Control key</b> and hold down the <b>left mouse</b> button. Drag the mouse button to create a rectangle that encloses an area. This area is then scaled to fit the drawing area.
<b>Reset View</b>	Select <b>View &gt; Reset</b> to reset the display to its initial state.
<b>Query</b>	Hold down the <b>left mouse</b> button near a graph marker. A window pops up displaying the dataset element's indices and value.

Figure 131 on page 261 shows a graph view of two dimensional random data created by selecting **Points** and deselecting **Lines** in the Data Window's **Graph Options** dialog box.

## Surface Window

The Surface Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (**x** and **y** axes) of the display. Figure 132 on page 261 shows a two-dimensional map, where the dataset values are shown using only the **Zone** option. (This demarcates ranges of element values.) For a zone map with contour lines, turn the **Zone** and **Contour** settings on and **Mesh** and **Shade** off.

You can display random data by selecting only the **Zone** setting and turning **Mesh**, **Shade**, and **Contour** off. The display shows where the data is located and allows you to click on it to get the values of the various points.

Figure 133 on page 262 shows a three-dimensional surface that maps element values to the height (**z** axis).

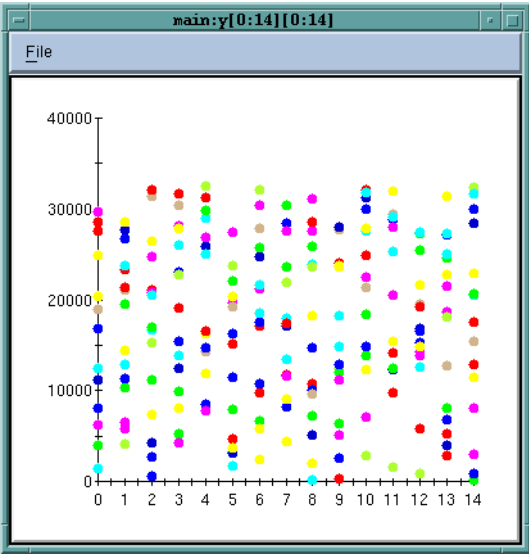


FIGURE 131: Display of Random Data

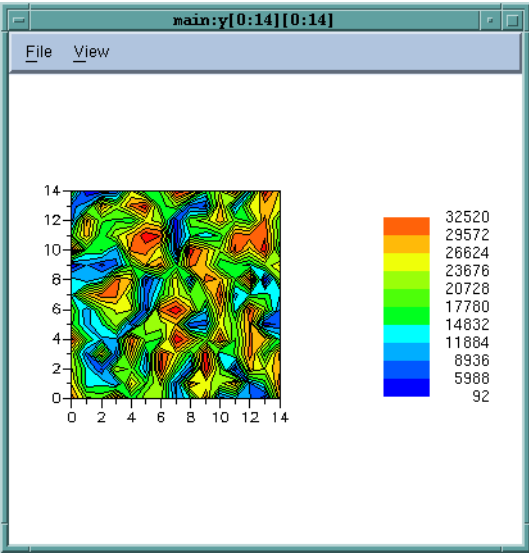


FIGURE 132: Two-Dimensional Surface Visualizer Data Display

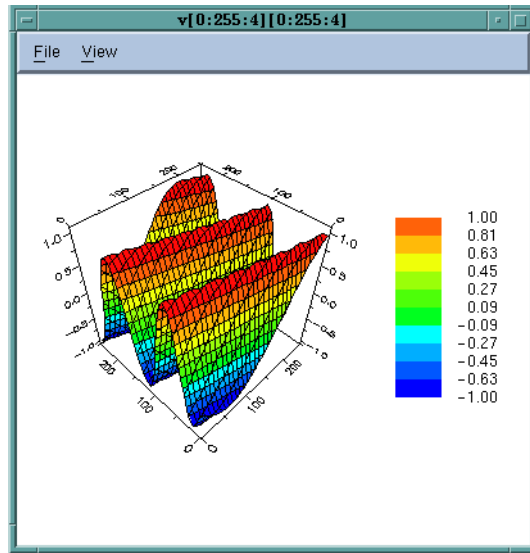


FIGURE 133: Three-Dimensional Surface Visualizer Data Display

### Displaying Surface Data

The Surface Window's **File > Options** command lets you control how the Visualizer displays the graph. (See Figure 134 on page 262.)

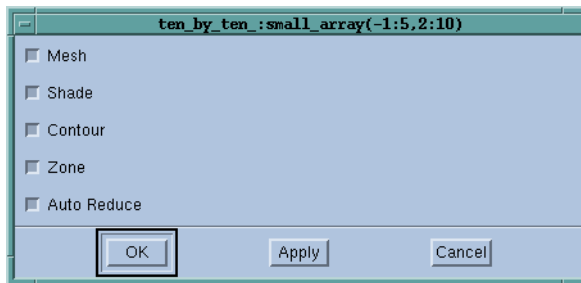


FIGURE 134: Surface Options Dialog Box

This dialog box has the following choices:

#### Mesh

If set, the Visualizer displays the surface as a three dimensional mesh, with the **x-y** grid projected onto the surface. When neither this option nor the **Shade** op-

	tion are set, the Visualizer displays the surface in two dimensions (See Figure 132).
<b>Shade</b>	If set, the Visualizer displays the surface in three dimensions and shaded either in a “flat” color to differentiate the top and bottom sides of the surface, or in colors corresponding to the value if the <b>Zone</b> option is also set. When neither this option nor the <b>Mesh</b> option are set, the Visualizer displays the surface in two dimensions. (See Figure 132 on page 261.)
<b>Contour</b>	If set, the Visualizer displays contour lines indicating ranges of element values.
<b>Zone</b>	If set, the Visualizer displays the surface in colors showing ranges of element values.
<b>Auto Reduce</b>	If set, the Visualizer derives the displayed surface by averaging over neighboring elements in the original dataset. This speeds up visualization by reducing the resolution of the surface. Clear this option if you want to accurately visualize all dataset elements.

The **Auto Reduce** option allows you to choose between viewing all your data points—which takes longer to appear in the display—or viewing the averaging of data over a number of nearby points.

You can reset the viewing parameters to those used when the Visualizer first came up by selecting the **View > Reset** command, which restores all translation, rotation, and scaling to its initial state and enlarges the display slightly.

## Manipulating Surface Data

The following commands change the display or give you information about it:

<b>Query</b>	Hold down the <b>left mouse</b> button near the surface. A window pops up displaying the nearest dataset element’s indices and value.
<b>Rotate</b>	Hold down the <b>middle mouse</b> button and <b>drag</b> the mouse to freely rotate the surface. You can also press the <b>x</b> , <b>y</b> , or <b>z</b> keys to select a single axis of rotation.

The Visualizer lets you rotate the surface in two dimensions simultaneously.

While you are rotating the surface, the Visualizer displays a wire-frame bounding box of the surface that moves with the mouse.

**Scale**

Press the **Control** key and hold down the **middle mouse** button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.

**Translate**

Press the **Shift** key and hold down the **middle mouse** button. Moving the mouse drags the surface.

**Zoom**

Press the **Control** key and hold down the **left mouse** button. Drag the mouse button to create a rectangle that encloses the area of interest. The Visualizer then translates and scales the area to fit the drawing area. See Figure 135.



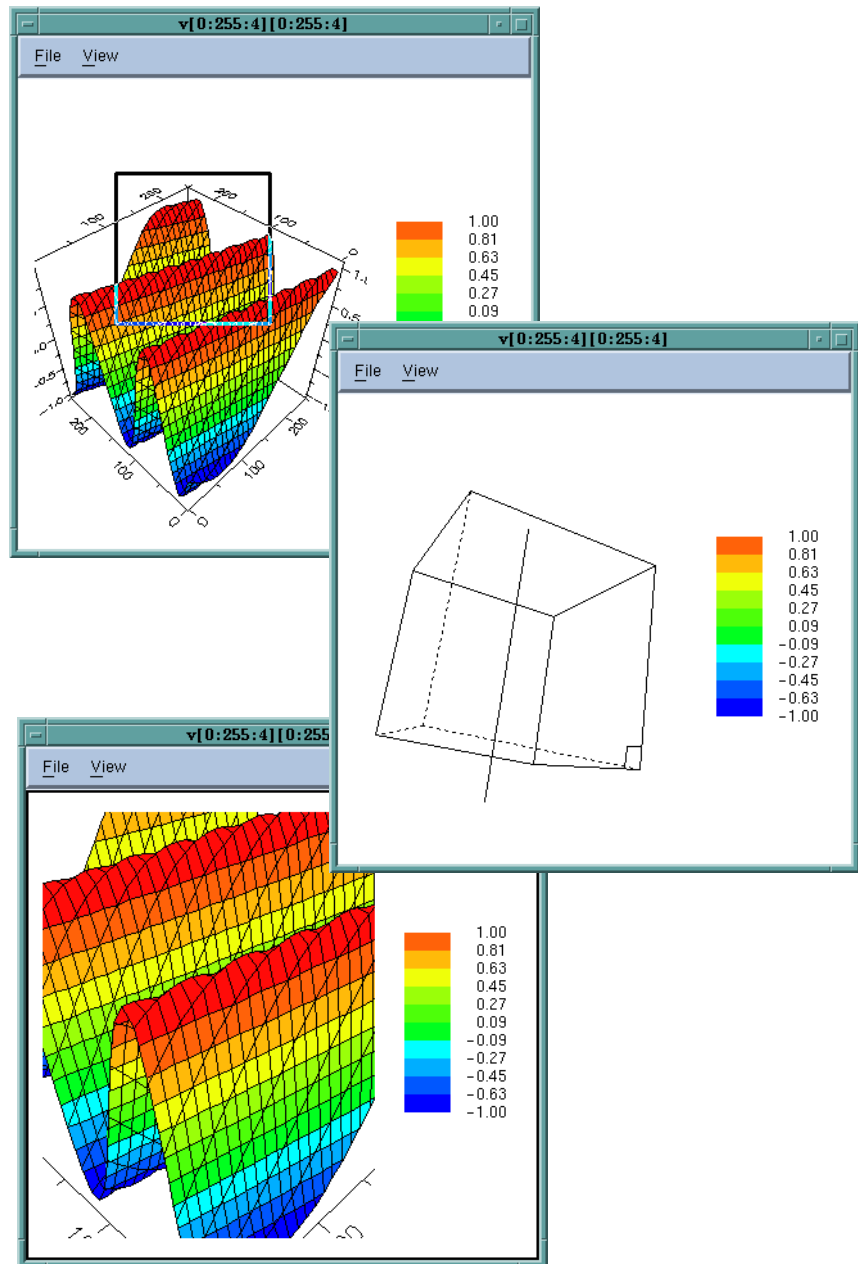


FIGURE 135: Zooming, Rotating, About an Axis

## Launching the Visualizer from the Command Line

To start the Visualizer from the shell, use the following syntax:

```
visualize [ -file filename | -persist ]
```

where:

- file** *filename*      Reads data from *filename* instead of reading from standard input.
- persist**              Continues to run after encountering an EOF on standard input. If you do not use this option, the Visualizer exits as soon as it reads all of the data from standard input.

By default, the Visualizer reads its input datasets from its standard input stream and exits when it reads an EOF on standard input. When started by TotalView, the Visualizer normally reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input from the standard input stream, invoke it by using the **-persist** option.

If you want to read data from a file, invoke the Visualizer with the **-file** option. For example:

```
visualize -file my_data_set_file
```

The Visualizer reads all the datasets in the file. This means that the images you see represent the last versions of the datasets in the file.

The Visualizer supports the generic X toolkit command-line options. For example, you can start the Visualizer with the Directory Window minimized by using the **-iconic** option. Your system manual page for the X server or the The “X Window System User’s Guide” by O’Reilly & Associates lists the generic X command-line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **-xrm resource\_setting** option. The available resources are described in Chapter 13 “TotalView Command Syntax” on page 289. Use of X resources to modify the default be-

havior of TotalView or the TotalView Visualizer is described in greater detail in Chapter 12 “X Resources” on page 275.



# Troubleshooting

This chapter describes how to solve common problems that you might encounter while using TotalView.

## Overview

This chapter discusses the following:

- Assembler is shown instead of source code
- Error creating new process
- Error launching process
- Fatal error: Checkout ... failed
- Fatal error in TotalView
- Hangs while debugging
- Internal error in TotalView
- Invalid license key
- License manager does not operate correctly
- Out-of-memory error
- Pressing Ctrl-C in an xterm window causes TotalView to exit
- Program behaves differently under TotalView control: setuid issues
- Program behaves differently under TotalView control: SIGSTOP problems
- Program symbols are not shown
- Single-stepping is slow or TotalView is slow to respond to breakpoints
- Source code does not appear in Source Pane
- TotalView cannot find your source code
- TotalView server (tvdsrv) fails to start on a remote node

- When debugging HPF programs, HPF source code does not appear in the Process Window; only f77 code appears
- Windows do not appear or operate correctly
- X resources are not recognized

The TOTALVIEW RELEASE NOTES contains extensive information on known problems. There you will find information on configuring TotalView, required operating patches, and workarounds.

If you cannot solve a problem, please contact us. You will find our bug reporting form in the support area of our web site and in our Release Notes. Or, you can phone us at 1-800-856-3766 in the United States or (+1) 508-875-3030 worldwide.

## The Problems

### *Assembler is shown instead of source code*

Check to make sure that you compiled your program using `-g`.

### *Error creating new process*

- Increase the swap space on your machine. For details, see "Swap Space" on page 323.
- Increase the number of process slots in your system. See your operating system documentation for details.
- Check the **xterm** window to see if the **execve()** call failed, and if it did, set the **PATH** environment variable.
- Make sure that the **/proc** file system is mounted on your system. For details, see "Mounting the /proc File System" on page 322.

### *Error launching process*

- Run your program from the UNIX command line prompt to see if it will load and start executing. (If it will not start from the UNIX command line, TotalView will not be able to start it.) If it does not run, make sure your program is built for the machine on which you are debugging. Or, an **execv()** system call fails because the file does not have execute permission. Or, maybe you are trying to run a 64-bit application on a machine that only runs 32-bit applications.

- Check that all shared libraries needed by your application are accessible. For example, you may not have properly set the dynamic library runtime loader path (which is **LD\_LIBRARY\_PATH** on most systems).
- Run your program from the UNIX command-line prompt to see if it will load and start executing. If it begins executing, you can start TotalView, and then attach to the executing program.
- TotalView cannot launch programs that are started by shell scripts. If it must be started by a shell script, you must manually start it and then attach to it from within TotalView.

#### *Fatal error: Checkout ... failed*

- Check the value of the **LM\_LICENSE\_FILE** environment variable. Make sure the value ends with the string **license.dat**. The default location for this file is in the **flexlm-6.1** subdirectory within your TotalView installation directory.
- Make sure the TotalView license manager **lmgrd** is running on the license manager host machine. The name of this machine is listed in the **SERVER** line of your **license.dat** file. The default location for this daemon is in the **flexlm-6.1/platform/bin** subdirectory within your TotalView installation directory.
- Make sure that the **lmgrd** that is running matches the one that came with your TotalView distribution. That is, if you are running other software that uses the FLEXlm license manager or if you have not upgraded an older version of FLEXlm, you might not be running the latest version.

#### *Fatal error in TotalView*

Report this problem. See “*Reporting Problems*” on page xvi.

#### *Hangs while debugging*

If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to be hung (it continuously displays the watch cursor). If you single-step a process over a statement that cannot complete without allowing another process to run and that process is stopped, the stepping process appears to hang. In parallel programs, this can occur when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by selecting yes from the question box that will appear. As an alternative, consider using a group-level single-step command instead.

### *Internal error in TotalView*

Report this problem. See “Reporting Problems” on page xvi.

### *Invalid license key*

Compare the format of your **license.dat** license key file with the one displayed in Chapter 2 of the TOTALVIEW INSTALLATION GUIDE. If you find stray characters in the file (for example “=3D”), use a text editor to remove them. After making these changes, stop the **lmgrd** license manager daemon and then restart it using the **toolworks\_init** script.

### *License manager does not operate correctly*

Set the **LM\_LICENSE\_FILE** environment variable to the path name of the TotalView license file. See the TOTALVIEW INSTALLATION GUIDE for details.

### *Out-of-memory error*

- Increase the swap space on your machine. For details, see “Swap Space” on page 323.
- Increase the data size limit in the C shell. Use the C shell’s **limit** command, such as:  
     % limit datasize unlimited

### *Pressing Ctrl-C in an xterm window causes TotalView to exit*

Start TotalView by using the **–ignore\_control\_c** command-line option.

### *Program behaves differently under TotalView control: setuid issues*

Make sure your program does not execute the **setuid** or **exec** functions or invoke another program that does, for example, **rsh**. Normally, the operating system does not allow a debugger to debug a **setuid** executable nor allow a **setuid** system call while a program is being debugged. Often these operations fail silently. To debug **setuid** programs, log in as the target UID before starting TotalView.

### *Program behaves differently under TotalView control: SIGSTOP problems*

TotalView uses the **SIGSTOP** signal to stop processes. On most UNIX systems, system calls can fail with *errno* set to **EINTR** when the process receives a **SIGSTOP** signal. You need to change your code so that it handles **EINTR** failures. For example:

```
do {
    n = read(fd,buf,nbytes);
} while (n < 0 && errno == EINTR);
```



When a system call is interrupted with a signal (for example, `errno == EINTR`), you need to retry it. This problem occurs because TotalView stops processes when it updates the displays. If the process is in a system call, the system call fails with `EINTR`.

For example, assume that your program has the following code fragment:

```
printf("creating scheduler thread...");
if (0 != (status = pthread_create(
    &scheduler_thread, &detached_attr,
    &scheduler_thread_wrapper, (void *)scheduler))) {
    error_func(ERR_LVL, __FILE__, __LINE__,
        "Pthread_create sScheduler, %d, %s",
        status, strerror(status));
}
```

You could restructure it to:

```
printf("creating scheduler thread...");
do {
    status = pthread_create(
        &scheduler_thread, &detached_attr,
        &scheduler_thread_wrapper, (void *)scheduler);
} while (0 != status && errno == EINTR);
if (0 != status) {
    error_func(ERR_LVL, __FILE__, __LINE__,
        "Pthread_create sScheduler, %d, %s",
        status, strerror(status));
}
```

### *Program symbols are not shown*

Check to make sure that you compiled your program using `-g`.

### *Single-stepping is slow or TotalView is slow to respond to breakpoints*

- Close some of the Variable Windows that you have open.
- The Globals Window is open and has a large number of variables. Close the Globals Window.
- If you set a breakpoint in a source file that has not yet been referenced or if you single-step into one, TotalView must read the file's symbol table. This can temporarily delay TotalView's response.

### *Source code does not appear in Source Pane*

- Set the search path for directories with the **File > Search Path** command in the Process Window.

- TotalView may be in the kernel or in a library routine for which source is not available.

#### *TotalView cannot find your source code*

Set the search path for directories with the **File > Search Path** command in the Process Window.

#### *TotalView server (tvdsrv) fails to start on a remote node*

Edit the Server Launch string within the **File > Preferences Launch Strings** Page, and relaunch the server. For information, see "Starting the TotalView Debugger Server" on page 61.

#### *When debugging HPF programs, HPF source code does not appear in the Process Window; only f77 code appears*

When compiling HPF programs, be sure to set the **-g** and **-Mtotalview** options when compiling and linking your programs.

#### *Windows do not appear or operate correctly*

- Your **DISPLAY** environment variable is not set correctly.
- The resource **totalview\*useTransientFor** on page 284 is not set correctly. Change it from **on** to **off**, or from **off** to **on**.
- Use the **xhost +** command to allow all hosts to access your display.

#### *X resources are not recognized*

- Use the **xrdb** command (part of the X Window System) to display the current X resources:  

```
xrdb -query
```
- Use the **xrdb** command to load your X resources:  

```
xrdb -load $HOME/.Xdefaults
```
- Read the **xrdb** manual page for more information.

# X Resources

This chapter provides reference information about the X Window System resources that you can use to customize TotalView or the TotalView Visualizer. You can use these resources in your X resources files (such as `.Xdefaults` on UNIX systems or `decw$sm_general.dat` on VMS systems).

For information on X resources files, refer to the X Window System documentation that came with your machine or the *X Window System User's Guide*, by O'Reilly & Associates (ISBN 1-56592-015-5).

On most UNIX systems, you load your X resources file by using the `xrdb` command (part of the X Window System executables). For example:

```
xrdb -load $HOME/.Xdefaults
```

Topics in this chapter are:

- TotalView X Resources
- Visualizer X Resources

## TotalView X Resources

You can override some of the resources with command-line options for the `totalview` command, as described in Chapter 13 “*TotalView Command Syntax*” on page 289.

**NOTE** You can specify an X resource on the command line by using the `-resource=value` command-line option. For example, to set “`totalview*stopAll`” to false, you could use the following command-line option: `-stopAll=false`. Note that the string “`totalview*`” is omitted from the command line.

**Window Locations:** Values for the location of windows are expressed as:

$$= \text{width} \times \text{height} + x + y$$

where *width* is the width of the window in pixels, *height* is the height of the window in pixels, *x* is the distance from the upper-left corner of the window to the left screen edge in pixels, and *y* is the distance from the upper-left corner of the window to the top screen edge in pixels. A value of -1 for *x* or *y* indicates that the window should be centered in the screen with respect to the x-axis or y-axis. If desired, you can express *x* or *y* as negative numbers to indicate the distance from the lower-right corner of the window to the bottom screen edge or right screen edge instead of the distance from the upper-left corner. A value of zero (0) indicates that TotalView should use the default value. Also, you can supply just the size (*width* and *height*), and TotalView will use the default location (*x* and *y*) with it.

As an example, the expression `=0x0-1+20` uses the default width and height, centers the window horizontally, and places the window 20 pixels down from the top of the screen. The expression `=330x120+20-20` makes the window 330 pixels wide by 120 pixels high and places the window 20 pixels from the left edge of the screen and 20 pixels up from the bottom edge of the screen.

**totalview\*autoRetraceAddresses:** {on | off}

If **on** (default), TotalView will retrace the sequence of dive operations performed in a Variable Window and recompute a new address for the variable. If **off**, TotalView does not retrace addresses.

**totalview\*backgroundColor:** *color*

Default: **white**

Sets the general background color to *color*.

**totalview\*compilerVars:** {true | false}

Compaq Tru64 UNIX and SGI only. If **false** (default), TotalView does not show variables created by the Fortran compiler. If **true**, TotalView shows variables created by the Fortran compiler and the variables in the user's program.

Some Fortran compilers (Compaq f90/f77 and SGI 7.2 compilers) output debug information that describes variables that the compiler itself has invented for purposes such as passing the length of **character\*(\*)** variables. By default, TotalView suppresses the display of these compiler-generated variables; however, setting **totalview\*compilerVars** to **true** tells TotalView to display these variables. This could be useful if you are looking for a corruption of a runtime descriptor or are writing a compiler.

Override with:    **-compiler\_vars** option (overrides **false**)  
                   **-no\_compiler\_vars** option (overrides **true**)

**totalview\*compileExpressions:** {**true** | **false**}

Compaq Alpha UNIX and IBM AIX (default **true**), and MIPS IRIX (default **false**) platforms only. If **true**, TotalView enables compiled expressions. If **false**, TotalView disables compiled expressions and interprets them instead.

**totalview\*cTypeStrings:** {**true** | **false**}

If **false** (default), TotalView's type string extensions are used when displaying the type strings for arrays. If **true**, C type string syntax are used when displaying arrays.

**totalview\*displayAssemblerSymbolically:** {**on** | **off**}

If **off** (default), display assembler locations as hexadecimal addresses. If **on**, assembler locations are displayed as "label+offset."

**totalview\*DPVMDebugging:** {**true** | **false**}

*Compaq Tru64 UNIX only.*

If **false** (default), disables support for debugging the Digital UNIX implementation of Parallel Virtual Machine (DPVM) applications. If **true**, enables support for debugging DPVM applications.

Override with:    **-dpvm** option (overrides **false**)  
                   **-no\_dpvm** option (overrides **true**)

**totalview\*font:** *fontname*

Default: **fixed**

Specifies the font used by the TotalView debugger. Use the X Windows-supplied application **xlsfonts** to list the names of available fonts.

**totalview\*foregroundColor:** *color*

Default: **black**

Sets the general foreground color (that is, the text color) to *color*.

**totalview\*globalTypenames:** {**true** | **false**}

If **true** (default), specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be **true** for standard conforming code.

If this option is **true**, TotalView attempts to replace an opaque type (**struct foo \*p;**) declared in one module with an identically named defined type (**struct foo { ... };**) in a different module.

If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable by using the non-opaque type definition.

If **false**, TotalView will not assume that type names are globally unique within a program. You should specify this option if your code has different definitions of the same named type since, otherwise, TotalView is likely to pick the wrong definition to substitute for an opaque type.

Override with:    **-global\_types** option (overrides **false**)  
                  **-no\_global\_types** option (overrides **true**)

**totalview\*hpf:** {**true** | **false**}

If **true** (default, if HPF debugging has been licensed), enables debugging at the HPF source level.

Setting this X resource to **false** causes TotalView to ignore **.stx** and **.stb** files, and therefore to debug HFP code at the intermediate (Fortran 77) level.

Override with:    **-hpf** option (overrides **false**)  
                   **-no\_hpf** option (overrides **true**)

**totalview\*hpfnNode:** {**true** | **false**}

If **false** (default), the node on which an HPF distributed array element resides is not displayed in the Process Window.

The node display can be toggled in each Variable Window by using the **Toggle Node Display** option in the Process Window menu.

Override with:    **-hpf\_node** option (overrides **false**)  
                   **-no\_hpf\_node** option (overrides **true**)

**totalview\*kccClasses:** {**true** | **false**}

If set to **true**, (default) TotalView will convert structure definitions output by the KCC compiler into classes that show base classes and virtual base classes in the same way as other C++ compilers. When set to **false**, the debugger will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers rather than as the data.

Unfortunately, the conversion has to be done by textual matching of the names given to structure members, so it can be confusing if you have structure component names that look to TotalView like KCC processed classes. However, the conversion is never performed unless TotalView believes that the code was compiled with KCC, because TotalView has seen one of the tag strings that KCC outputs, or because the user has asked for the KCC name demangler to be used. Also all of the recognized structure component names start with "**\_\_**", and, according to the C standard, user code should not contain names with this prefix.

Note that under some circumstances it is not possible to convert the original type names because there is no available type definition. For example, it may not be possible to convert "**struct \_\_SO\_foo**" to "**struct foo**", so in this case the "**\_\_SO\_foo**" type will be shown. This is only a cosmetic prob-

lem. (The “\_\_SO\_\_” prefix denotes a type definition for the nonvirtual components of a class with virtual bases).

Since KCC outputs no information on the accessibility of base classes (“private”, “protected”, “public”), TotalView is unable to provide this information.

**totalview\*parallelAttach:** { yes | no | ask }

Tells TotalView what it should do when it can automatically attach to processes. Your choices are as follows:

- **yes:** Attach to all started processes.
- **no:** Do not attach to any started processes.
- **ask:** Display a dialog box listing the processes to which TotalView can attach and let the user decide which ones TotalView should attach to.

**totalview\*parallelStop:** { yes | no | ask }

Tells TotalView if it should automatically run processes when your program launches them. Your choices are as follows:

- **yes:** Stop the processes before they begin executing.
- **no:** Do not interfere with the processes; that is, let them run.
- **ask:** Display a question box asking what it should do.

**totalview\*patchAreaAddress:** *address*

Allocates the patch space dynamically at the given *address*. See “*Allocating Patch Space for Compiled Expressions*” on page 221.

**totalview\*patchAreaLength:** *length*

Sets the length of the dynamically allocated patch space to the specified *length*. See “*Allocating Patch Space for Compiled Expressions*” on page 221.

**totalview\*popAtBreakpoint:** { on | off }

If **on**, sets the **Open (or raise) process window at breakpoint** check box to be selected by default. If **off** (default), sets that check box to be deselected by default. See “*Handling Signals*” on page 45.



Override with:    –pop\_at\_breakpoint option (overrides **off**)  
                   –no\_pop\_at\_breakpoint option (overrides **on**)

### totalview\*popOnError: {on | off}

If **on** (default), sets the **Open process window on error signal** check box within the **File > Preference's** Option Page to be selected by default. If **off**, sets that check box to be deselected by default. See “*Handling Signals*” on page 45 for more information.

Override with:    –pop\_on\_error option (overrides **off**)  
                   –no\_pop\_on\_error option (overrides **on**)

### totalview\*pvmDebugging: {true | false}

If **false** (default), disables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications. If **true**, enables support for debugging PVM applications.

Override with:    –pvm option (overrides **false**)  
                   –nopvm option (overrides **true**)

### totalview\*searchCaseSensitive: {on | off}

If **off** (default), searching for strings is not case sensitive. If **on**, searches are case- sensitive.

### totalview\*searchPath: dir1[,dir2,...]

Specifies a list of directories for the debugger to search when looking for source and object files. This resource serves the same purpose as the **File > Search Path** command in the Process Window (see “*Setting Search Paths*” on page 48). If you use multiple lines, place a backslash (\) at the end of each line, except for the last line.

### totalview\*signalHandlingMode: action\_list

Modifies the way in which TotalView handles signals. An *action\_list* consists of a list of *signal\_action* descriptions, separated by spaces:

*signal\_action*[*signal\_action*] ...

A *signal\_action* description consists of an action, an equal sign (=), and a list of signals:

*action=signal\_list*

An *action* can be one of the following: **Error**, **Stop**, **Resend**, or **Discard**. For more information on the meaning of each action, refer to “Handling Signals” on page 45.

A *signal\_list* is a list of one or more signal specifiers, separated by commas:

*signal\_specifier[,signal\_specifier] ...*

A *signal\_specifier* can be a signal name (such as **SIGSEGV**), a signal number (such as **11**), or a star (\*), which specifies all signals. We recommend using the signal name rather than the number because number assignments vary across UNIX versions.

The following rules apply when you are specifying an *action\_list*:

- If you specify an action for a signal in an *action\_list*, TotalView changes the default action for that signal.
- If you do not specify a signal in the *action\_list*, TotalView does not change its default action for the signal.
- If you specify a signal that does not exist for the platform, TotalView ignores it.
- If you specify an action for a signal twice, TotalView uses the last action specified. In other words, TotalView applies the actions from left to right.

If you need to revert the settings for signal handling to TotalView’s built-in defaults, use the **Defaults** button in the **File > Signals** dialog box.

For example, to set the default action for the **SIGTERM** signal to *Resend*, you specify the following action list:

**“Resend=SIGTERM”**

As another example, to set the action for **SIGSEGV** and **SIGBUS** to *Error*, the action for **SIGHUP** and **SIGTERM** to *Resend*, and all remaining signals to *Stop*, you specify the following action list:

**“Stop=\* Error=SIGSEGV,SIGBUS Resend=SIGHUP,SIGTERM”**

This action list shows how TotalView applies the actions from left to right.

- 1 Sets the action for all signals to *Stop*.
- 2 Changes the action for **SIGSEGV** and **SIGBUS** from *Stop* to *Error*.
- 3 Changes the action for **SIGHUP** and **SIGTERM** from *Stop* to *Resend*.

**totalview\*sourcePaneTabWidth:** *n*

Default: **8**

Sets the width of the tab character that is displayed in the Source Pane. For example, if your source file uses a tab width of 4, set *n* to **4**.

**totalview\*spellCorrection:** {**verbose** | **brief** | **none**}

When you use the **Function or File...** or **Variable...** commands in the Process Window or edit a type string in a Variable Window, the debugger checks the spelling of your entries. By default (**verbose**), the debugger displays a dialog box before it corrects spelling. You can set this resource to **brief** to run the spelling corrector silently. (The debugger makes the spelling correction without displaying it in a dialog box first.) You can also set this resource to **none** to disable the spelling corrector.

**totalview\*useInterface:** *name*

Sets the interface name that the server uses when it makes a callback. For example, on an IBM PS2 machine, the following resource setting sets the callback to use the hardware option:

**totalview\*useInterface:css0**

However, TotalView will let you use any legal **inet** interface name. (You can obtain a list of the interfaces if you use the **netstat -i** command.)

**totalview\*userThreads:** {**true** | **false**}

If set to **true** (default), enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported. If set to **false**, TotalView disables handling of user-level (M:N) thread packages. Disabling thread support may be useful in situations

where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.

*Override with:*    **–user\_threads** option (overrides **false**)  
                       **–no\_user\_threads** option (overrides **true**)

**totalview\*useTransientFor:** {**on** | **off**}

If **off**, TotalView sets the “override redirect” property for windows. This property does not let you use the window manager to perform operations such as raising and lowering dialog boxes. If you use an advanced window manager, you can use the **on** option (default) which lets TotalView use “transient-for” type windows. This property allows you to use the window manager to perform operations on dialog boxes. If you are using an X11R4 or more recent server and window manager, you should use the **on** option. If you’re using Compaq’s window manager, you should use the **off** option.

**totalview\*verbosity:** {**silent** | **error** | **warning** | **info**}

*Default:* **info**

Sets the verbosity level of TotalView-generated messages.

## Visualizer X Resources

The TotalView Visualizer uses a large number of X resources that are set up in its application defaults file. The X resources documented are a subset of those found in the application defaults file as they are the only ones that can be customized to your preferences. Setting them in your own X resources file overrides the application defaults file.

The default values of the X resources are listed here shown either in a bold typeface in a list of alternative values, or separately if there can be a range of values. They are the settings in the applications defaults file as it is shipped. Your site administrator can edit this file to set the site defaults; therefore, your site may have different defaults.

**Visualize\*data\*pick\_message.background:** *color*

*Default:* **light yellow**

Sets the color of the pick pop-up window.

**Visualize\*directory\*auto\_visualize.set:** {1 | 0}

Sets the initial state of the autovisualize option in the Directory Window. If set (1), when a new dataset is added to the list, it will be visualized automatically using an appropriate method. If cleared (0), the new dataset will not be displayed automatically, and you will have to choose a visualization method for it.

**Visualize\*directory.height:** *height*

*Default:* **100**

Sets the initial height of the Directory Window.

**Visualize\*directory.width:** *width*

*Default:* **300**

Sets the initial width of the Directory Window.

**Visualize\*graph.height:** *height*

*Default:* **400**

Sets the initial height of the Graph Window.

**Visualize\*graph.width:** *width*

*Default:* **400**

Sets the initial width of the Graph Window.

**Visualize\*graph\*lines.set:** {1 | 0}

Sets the initial state of the lines option in the Graph Window. When set (1), graphs are drawn with lines connecting the data points.

**Visualize\*graph\*points.set:** {1 | 0}

Sets the initial state of the points option in the Graph Window. When set (1), graphs are drawn with markers on each data point.

**Visualize\*surface.height:** *height*

Default: 400

Sets the initial height of the Surface Window.

**Visualize\*surface.width:** *width*

Default: 400

Sets the initial width of the Surface Window.

**Visualize\*surface\*mesh.set:** {1 | 0}

Sets the initial state of the mesh option in the Surface Window. When set (1), the axis grid is projected onto the surface.

**Visualize\*surface\*shade.set:** {1 | 0}

Sets the initial state of the shade option in the Surface Window. When set (1), the surface is shaded.

**Visualize\*surface\*contour.set:** {1 | 0}

Sets the initial state of the contour option in the Surface Window. When set (1), contours are displayed on the surface.

**Visualize\*surface\*auto\_reduce.set:** {1 | 0}

Sets the initial state of the autoreduce option in the Surface Window. When set (1), large datasets are reduced by averaging to speed display.

**Visualize\*surface\*xrt3dZoneMethod:** {zonecontours | zonecells}

Specifies how the surface is colored. When set to **zonecontours**, the surface is colored according to its contours. When set to **zonecells**, each cell in the mesh is colored based on the average value in the cell.

**Visualize\*surface\*xrt3dViewNormalized:** {1 | 0}

When set (1), the view of the dataset (before zooming or translation) is maximized to fit the window. Interactive rotation when this resource is set will look “jerky” but will ensure no portion of the display is clipped. When this resource is cleared (0), dynamic rotation will be smooth, but parts of the display (for example, axes) may be clipped at some viewing angles.

**Visualize\*surface\*xrt3dXMeshFilter:**  $n$ 

*Default:* 0

Specifies how to display the surface mesh. Every  $n$ th mesh line will be displayed, where  $n$  must be an integer greater than or equal to 0. When set to 0, a value is calculated automatically.

**Visualize\*surface\*xrt3dYMeshFilter:**  $n$ 

*Default:* 0

Specifies how to display the surface mesh. Every  $n$ th mesh line will be displayed, where  $n$  must be an integer greater than or equal to 0. When set to 0, a value is calculated automatically.

**Visualize\*surface\*zone.set:** {1 | 0}

Sets the initial state of the zone option in the Surface Window. When set (1), the surface is colored according to the value.





# TotalView Command Syntax

This chapter describes the syntax of the `totalview` command. Topics in this chapter are:

- Syntax
- Options

## Syntax

**Synopsis:** `totalview [ filename [ corefile ] ] [ options ]`

**Description:** The TotalView debugger is a source-level debugger with a graphic interface (based on the X Window System) and features for debugging distributed programs, multiprocess programs, and multithreaded programs. You need a workstation or terminal running the X Window System to use TotalView. TotalView is available on a number of different platforms.

### Arguments:

<i>filename</i>	Specifies the path name of the executable being debugged. This can be an absolute or relative path name. The executable must be compiled with debugging symbols turned on, normally the <code>-g</code> compiler option. Any multiprocess programs that call <code>fork()</code> , <code>vfork()</code> , or <code>execve()</code> should be linked with the <code>dbfork</code> library.
-----------------	---

<i>corefile</i>	Specifies the name of a core file. Specify this argument in addition to <i>filename</i> when you want to examine a core file with TotalView:
-----------------	--

`totalview filename corefile [ options ]`

.....

–a *args*

**Using Options :** If you specify mutually exclusive options on the same command line (for example, **–dynamic** and **–no\_dynamic**), the last option listed is used. Some of these options override TotalView X resources described in “X Resources” on page 275. If an option contains underscores (**\_**), you can usually omit the underscores. For example, **–nodynamic** is the same as **–no\_dynamic**; similarly **–arrowbgcolor** and **–arrow\_bg\_color** are the same.

**NOTE** The option **–Xresource=value** option allows you to set the X resource *Xresource* to *value* from the command line. For example, to set “**totalview\*stopAll**” to **false**, you could specify the command-line option **–stopAll=false**. Note that the string “**totalview\***” is omitted from the command line. X resource values set from the command line override settings in your X resource file. For a complete list of X resources, see Chapter 12 “X Resources” on page 275.

## Options

.....

- |                          |   |
|--------------------------|---|
| –a <i>args</i>           | Passes all subsequent arguments (specified by <i>args</i> ) to the program specified by <i>filename</i> . This option must be the <i>last</i> one on the command line.  |
| –ask_on_dlopen           | (Default) TotalView will ask you about stopping processes that dynamically load a new shared library by using the <b>dlopen</b> or <b>load</b> (AIX only) system calls. See “ <i>Debugging Dynamically Loaded Libraries</i> ” on page 331.  |
| –no_ask_on_dlopen        | TotalView will <i>not</i> ask you about stopping processes that dynamically load a new shared library by using the <b>dlopen</b> or <b>load</b> (AIX only) system calls. See “ <i>Debugging Dynamically Loaded Libraries</i> ” on page 331. |
| –background <i>color</i> | Sets the general background color to <i>color</i> .<br>Default: <b>white</b>  |
| –bg <i>color</i>         | Same as <b>–background</b> .  |
| –barr_stop_all           | (Default) Enables process barrier breakpoints to stop all related processes.  |

–bulk\_launch\_tmpfile1\_trailer\_string *string*

–no\_barr\_stop\_all

The process barrier breakpoint does *not* stop all related processes.

–bulk\_launch\_base\_timeout *time\_in\_seconds*

Sets the time to wait before giving up trying to establish the connections. The total timeout is calculated as a base value (this option), in addition to an amount for each server launched. That time is specified using the –bulk\_launch\_incr\_timeout value.

–bulk\_launch\_incr\_timeout *time\_in\_seconds*

Sets the time to wait before giving up trying to establish the connections. The total timeout is calculated as a base value (indicated with the –bulk\_launch\_base\_timeout command) and an amount for each server launched (this option).

–bulk\_launch\_string *string*

Defines the launch string used to launch the bulk server. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile1\_header *string*

Defines the first temporary header file that will be created during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile1\_host\_string *string*

Defines the first host string sent to the remote process during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile1\_trailer\_string *string*

Defines the first temporary trailer file that will be created during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile2\_header\_string *string*

–bulk\_launch\_tmpfile2\_header\_string *string*

Defines the second temporary header file that will be created during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile2\_host\_string *string*

Defines the second host string sent to the remote process during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–bulk\_launch\_tmpfile2\_trailer\_string *string*

Defines the second temporary trailer file that will be created during a bulk server launch. See the **Bulk Launch** Page within the **File > Preferences** dialog box for more information.

–compiler\_vars

(Alpha, HP, and SGI only.) Shows variables created by the Fortran compiler, as well as those in the user's program.

–no\_compiler\_vars

(Default) Does not show variables created by the Fortran compiler.

Some Fortran compilers (Compaq f90/f77, HP f90, SGI 7.2 compilers) output debug information that describes variables the compiler itself has invented for purposes such as passing the length of character\*(\*) variables. By default, TotalView suppresses the display of these compiler-generated variables.

However, you can specify the **–compiler\_vars** option or set the **totalview\*compilerVars** X resource to **true** to cause such variables to be displayed. This could be useful if you are looking for a corruption of a runtime descriptor or are writing a compiler.

–dbfork

(Default) Catches the **fork()**, **vfork()**, and **execve()** system calls if your executable is linked with the **dbfork** library.

**-no\_dbfork** Does not catch `fork()`, `vfork()`, and `execve()` system calls even if your executable is linked with the **dbfork** library.

**-debug\_file** *consoleoutputfile*

Redirects TotalView console output to a file named *consoleoutputfile*.

*Default:* All TotalView console output is written to **stderr**.

**-demangler=***compiler*

Overrides the C++ demangler and mangler TotalView uses by default. The following table lists override options.

TABLE 22: Demangling Command-Line Options

Option	Meaning
<b>-demangler=compaq</b>	Compaq cxx on Linux (alpha)
<b>-demangler=cset</b>	IBM CSet ++
<b>-demangler=dec</b>	Compaq Tru64 C++
<b>-demangler=gnu</b>	GNU C++
<b>-demangler=hp</b>	HP aCC compiler
<b>-demangler=irix</b>	SGI IRIX C++
<b>-demangler=kai</b>	KAI C++
<b>-demangler=kai3_n</b>	KAI C++ version 3.n
<b>-demangler=spro</b>	SunPro C++ 4.0 or 4.2
<b>-demangler=spro5</b>	SunPro C++ 5.0 or later
<b>-demangler=sun</b>	Sun C++

**-display** *displayname*

Sets the name of the X Windows display to *displayname*. For example, **-display vinnie:0.0** will display TotalView on the machine named "vinnie."

*Default:* To the value of the **DISPLAY** environment variable.

.....  
`-dll_ignore_prefix list`

`-dll_ignore_prefix list`

The colon-separated argument to this option tells TotalView that it should ignore files having this prefix when making a decision to ask about stopping the process when it *dllopens* a dynamic library. If the DLL being opened has any of the entries on this list as a prefix, the question is not asked.

`-dll_stop_suffix list`

The colon-separated argument to this option tells TotalView that if the library being opened has any of the entries on this list as a suffix, it should ask if it should open the library.

`-dpvm`

Compaq Tru64 UNIX only: Enables support for debugging the Compaq Tru64 UNIX implementation of Parallel Virtual Machine (PVM) applications.

`-no_dpvm`

Compaq Tru64 UNIX only: (Default) Disables support for debugging the Compaq Tru64 UNIX implementation of PVM applications.

`-dump_core`

Allows TotalView to dump a core file when it gets an internal error. Useful for debugging TotalView itself.

`-no_dumpcore`

(Default) Does not allow TotalView to dump a core file when it gets an internal error.

`-dynamic`

(Default) Loads symbols from shared libraries. This option is available only on platforms that support shared libraries.

`-no_dynamic`

Does not load symbols from shared libraries when reading dynamically linked executables. Setting this option can cause the **dbfork** library to fail because TotalView might not find the **fork()**, **vfork()**, and **execve()** system calls.

`-editor_launch_string string`

Defines the launch string used to launch a text editor when you select the Process Window's **File > Edit Source** command. See the **Launch Strings** page within the **File > Preferences** dialog box for more information.

- ext** *extension* Specifies that files with the suffix *extension* are preprocessor input files. TotalView already has built-in extensions for C++ (.C, .cpp, .cc, .cxx), Fortran (.F), **lex** (.l, .lex), and **yacc** (.y) files.
- fixed\_font\_family** *fontname* Specifies the fixed-width font that TotalView uses when displaying your source code and other similar information. Use the **-fixed\_font\_size** option to specify the size at which this font is displayed.
- It is usually better to specify this family by using a TotalView preference.
- fixed\_font\_size** *number* Specifies the size at which TotalView displays the font indicated with **-fixed\_font\_family** option.
- It is usually better to specify this size by using a TotalView preference.
- foreground** *color* Sets the general foreground color (that is, the text color) to *color*.
- Default: **black**
- fg** *color* Same as **-foreground**.
- global\_types** (Default) Specifies that TotalView can assume that type names are globally unique within a program and that all type definitions with the same name are identical. In C++, the standard asserts that this must be true for standard conforming code.
- If this option is set, TotalView will attempt to replace an opaque type (**struct foo \*p;**) declared in one module, with an identically named defined type in a different module.
- If TotalView has read the symbols for the module containing the non-opaque type definition, then when displaying variables declared with the opaque type, TotalView will automatically display the variable by using the non-opaque type definition.

### ..... -no\_global\_types

- no\_global\_types Specifies that TotalView *cannot* assume that type names are globally unique within a program. You should specify this option if your code has multiple different definitions of the same named type, since otherwise TotalView is likely to pick the wrong definition to substitute for an opaque type.
  
- hpf (Default) Enables debugging HPF code at the source level.
- no\_hpf Disables debugging HPF source code at the source level.
  
- hpf\_node Enables display of the node on which the HPF distributed array element resides in the Process Window.
- no\_hpf\_node (Default) Disables display of the node on which the HPF distributed array element resides in the Process Window.
  
- ignore\_control\_c Ignores Ctrl-C and prevents you from terminating the TotalView process from an **xterm** window, which is useful when your program catches the Ctrl-C signal (**SIGINT**).
  
- icc Same as -ignore\_control\_c.
- no\_ignore\_control\_c (Default) Catches Ctrl-C and terminates your TotalView debugging session. To override this, use -ignore\_control\_c.
- nicc Same as -no\_ignore\_control\_c.
  
- kcc\_classes (Default) Converts structure definitions output by the KCC compiler into classes that show base classes, and virtual base classes in the same way as other C++ compilers. See the description of the X resource **totalview\*kccClasses** on page 279 for a description of the conversion performed by TotalView.
- no\_kcc\_classes Specifies that TotalView will not convert structure definitions output by the KCC compiler into classes. Virtual bases will show up as pointers, rather than the data.



- lb (Default) Loads action points automatically from the *filename.TVD.breakpoints* file, providing the file exists.
- nlb Does not load action points automatically from an action points file.
- message\_queue (Default) Enables the display of MPI message queues when debugging an MPI program.
- mqd Same as -message\_queue.
- no\_message\_queue Disables the display of MPI message queues when debugging an MPI program. This might be useful if a store corruption is overwriting the message queues and causing TotalView to become confused.
- no\_mqd Same as -no\_message\_queue.
- parallel (Default) Enables handling of parallel program runtime libraries such as MPI, PE, and HPF.
- no\_parallel Disables handling of parallel program runtime libraries such as MPI, PE, and HPF. This is useful for debugging parallel programs as if they were single-process programs.
- parallel\_attach *argument* Tells TotalView what it should do when automatically attaching to processes. The values you can enter are:
  - yes** (attach to all started processes)
  - no** (do not attach to any started process)
  - ask** (display a dialog box listing the processes and let the user decide which ones TotalView should attach to)
- parallel\_stop *argument* Tells TotalView if it should automatically run processes when your program launches them. The values you can enter are:
  - yes** (stop the processes before they begin executing)
  - no** (do not interfere with the processes; that is, let them run)
  - ask** (display a question box asking what it should do)

.....  
 -patch\_area\_base address

-patch\_area\_base *address*

Allocates the patch space dynamically at the given *address*. See "Allocating Patch Space for Compiled Expressions" on page 221.

-patch\_area\_length *length*

Sets the length of the dynamically allocated patch space to the specified *length*. See "Allocating Patch Space for Compiled Expressions" on page 221.

-pop\_at\_breakpoint

Sets the **Open (or raise) process window at breakpoint** check box to be selected by default. See "Handling Signals" on page 45.

-no\_pop\_at\_breakpoint

(Default) Sets the **Open (or raise) process window at breakpoint** check box to be deselected by default.

-pop\_on\_error

(Default) Sets the **Open (or raise) process window on error** check box to be selected by default. See "Handling Signals" on page 45.

-no\_pop\_on\_error

Sets the **Open (or raise) process window on error** check box to be deselected by default.

-pvm

Enables support for debugging the ORNL implementation of Parallel Virtual Machine (PVM) applications.

-no\_pvm

(Default) Disables support for debugging the ORNL implementation of PVM applications.

-remote *hostname[:portnumber]*

Debugs an executable that is not running on the same machine as TotalView. For *hostname*, you can specify a TCP/IP host name (such as **vinnie**) or a TCP/IP address (such as **128.89.0.16**). Optionally, you can specify a TCP/IP port number for *portnumber*, such as **:4174**. When you specify a port number, you disable the auto-launch feature. For more information on the auto-launch feature, see "Single Process Server Launch Command" on page 66.

- r** *hostname[:portnumber]*  
Same as **-remote**.
- s** *pathname*  
Specifies the path name of a startup file that will be loaded and executed. This path name can either be an absolute or relative name. You can find information on the contents of this start-up file in the CLI GUIDE.
- sb**  
Saves action points automatically to an action points file when you exit TotalView. The file is named *filename.TVD.breakpoints*.
- nsb**  
(Default) Does not save action points automatically to an action points file when you exit.
- serial** *device[:options]*  
Debugs an executable that is not running on the same machine as TotalView. For *device*, specify the device name of a serial line, such as **/dev/com1**. Currently, the only *option* you are allowed to specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see "Debugging Over a Serial Line" on page 72.
- server\_launch\_string** *string*  
Defines the launch string used to launch a remote server. See the **Launch Strings** Page within the **File > Preferences** dialog box for more information.
- signal\_handling\_mode** "*action\_list*"  
Modifies the way in which TotalView handles signals. You must enclose the *action\_list* string in quotation marks to protect it from the shell. Refer to **totalview\*signalHandlingMode** on page 281 for a description of the *action\_list* argument.
- shm** "*action\_list*"  
Same as **-signal\_handling\_mode**.
- stop\_all**  
(Default) Sets the **Stop All Related Processes when Breakpoint Hit** check box to be selected by default. To override this option use **-no\_stop\_all**. See "Breakpoints for Multiple Processes" on page 209.

.....  
`-ui_font_family` *fontname*

`-no_stop_all` Sets the **Stop All Related Processes when Breakpoint Hit** check box to be deselected by default.

`-ui_font_family` *fontname*

Specifies the variable-width font that TotalView uses. Use the `-ui_font_size` option to specify the size at which this font is displayed.

It is usually better to specify this family by using a TotalView preference.

`-ui_font_size` *number*

Specifies the size at which TotalView displays the font indicated with `-ui_font_family` option.

It is usually better to specify this size using a TotalView preference.

`-user_threads`

(Default) Enables handling of user-level (M:N) thread packages on systems where two-level (kernel and user) thread scheduling is supported.

`-no_user_threads`

Disables handling of user-level (M:N) thread packages. This option may be useful in situations where you need to debug kernel-level threads, but in most cases, this option is of little use on systems where two-level thread scheduling is used.

`-verbosity` *level*

Sets the verbosity level of TotalView-generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

`-visualizer_launch_string` *string*

Defines the launch string used to launch a visualizer when you select the **Tools > Visualize** command within the Variable Window or when TotalView encounters a **\$visualize** intrinsic when it is evaluating an expression. See the **Launch Strings** Page within the **File > Preferences** dialog box for more information.

—visualizer\_max\_rank *number*

—visualizer\_max\_rank *number*

Specifies the number of array dimensions that are sent to a visualizer. If you are using TotalView's Visualizer, this value cannot be greater than 2. The maximum value you can specify is 16.

.....  
-visualizer\_max\_rank number

# TotalView Debugger Server (tvdsvr) Command Syntax

This chapter summarizes the syntax of the TotalView Debugger Server command, **tvdsvr**, which is used for remote debugging. For more information on remote debugging, refer to “*Starting the TotalView Debugger Server*” on page 61.

Topics in this chapter are:

- The **tvdsvr** Command and Its Options
- Replacement Characters

## The tvdsvr Command and Its Options

**Synopsis:** `tvdsvr {-server | -callback hostname:port | -serial device}  
[other options]`

**Description:** The **tvdsvr** debugger server allows TotalView to control and debug a program on a remote machine. To accomplish this, the **tvdsvr** program must run on the remote machine, and it must have access to the executables to be debugged. These executables must have the same absolute path name as the executable that TotalView is debugging, or the **PATH** environment variable for **tvdsvr** must include the directories containing the executables.

You must specify either the **-server**, **-callback**, or **-serial** option with the **tvdsvr** command. By default, the TotalView debugger automatically

.....  
**-callback** *hostname:port*

launches **tvdsvr** (known as the autolaunch feature) with the **-callback** option, and the server establishes a connection with TotalView.

If you prefer not to use the autolaunch feature, you can start **tvdsvr** manually and specify the **-server** option. Be sure to note the password that **tvdsvr** prints out with the message:

**pw** = *hexnumhigh:hexnumlow*

TotalView will prompt you for *hexnumhigh:hexnumlow* later. By default, **tvdsvr** automatically generates a password that is used when establishing connections. If desired, you can use the **-set\_pw** option to set a specific password.

To connect to the **tvdsvr** from TotalView, you use the **Fille > New Program** dialog box and must specify the host name and TCP/IP port number, *hostname:portnumber* on which **tvdsvr** is running. Then, TotalView prompts you for the password for **tvdsvr**.

**Options:** The following options determine the port number and password necessary for TotalView to connect with **tvdsvr**.

**-callback** *hostname:port*

(Autolaunch feature only) Immediately establishes a connection with a TotalView process running on *hostname* and listening on *port*, where *hostname* is either a host name or TCP/IP address. If **tvdsvr** cannot connect with TotalView, it exits.

If you use the **-port**, **-search\_port**, or **-server** options with this option, **tvdsvr** ignores them.

**-callback\_host** *hostname*

Names the host upon which the callback is made. *hostname* indicates the machine upon which TotalView is running. This option is most often used with a bulk launch.

**-callback\_ports** *port-list*

Names the ports on the host machines that are used for callbacks. The *port-list* argument contains a



comma-separated list of the host names and TCP/IP port numbers (*hostname:port,hostname:port...*) on which TotalView is listening for connections from **tvdsvr**. This option is most often used with a bulk launch.

**-debug\_file** *consoleoutputfile*

Redirects TotalView Debugger Server console output to a file named *consoleoutputfile*.

*Default:* All console output is written to **stderr**.

**-dpvm**

Uses the Compaq Tru64 UNIX implementation of the Parallel Virtual Machine (DPVM) library process as its input channel and registers itself as the DPVM tasker.

**NOTE** This option is not intended for users launching **tvdsvr** manually. When you enable DPVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

**-port** *number*

Sets the TCP/IP port number on which **tvdsvr** should communicate with **totalview**. If this TCP/IP port number is busy, **tvdsvr** does not select an alternate port number (that is, it communicates with nothing) unless you also specify **-search\_port**.

*Default:* 4142

**-pvm**

Uses the ORNL implementation of the Parallel Virtual Machine (PVM) library process as its input channel and registers itself as the ORNL PVM tasker.

**NOTE** This option is not intended for users launching **tvdsvr** manually. When you enable PVM support within TotalView, TotalView automatically uses this option when it launches **tvdsvr**.

**-search\_port**

Searches for an available TCP/IP port number, beginning with the default port (4142) or the port set with the **-port** option and continuing until one is found. When the port number is set, **tvdsvr** displays the chosen port number with the following message:

**port** = *number*

—serial *device[:options]*

Be sure that you remember this port number, since you will need it when you are connecting to this server from TotalView.

—serial *device[:options]*

Waits for a serial line connection from TotalView. For *device*, specify the device name of a serial line, such as */dev/com1*. The only *option* you can specify is the baud rate, which defaults to **38400**. For more information on debugging over a serial line, see “*Debugging Over a Serial Line*” on page 72.

—server

Listens for and accepts network connections on port 4142 (default).

Using **—server** can be a security problem. Consequently, you must explicitly enable this feature by placing an empty file named **tvdsvr.conf** in your **/etc** directory. This file must be owned by user ID 0 (root). When **tvdsvr** encounters this option, it checks if this file exists. This file’s contents are ignored.

You can use a different port by specifying either **—port** or **—search\_port**. To stop **tvdsvr** from listening and accepting network connections, you must terminate it by pressing Ctrl-C in the terminal window from which it was started or by using the **kill** command.

—set\_pw *hexnumhigh:hexnumlow*

Sets the password to the 64-bit number specified by the two 32-bit numbers *hexnumhigh* and *hexnumlow*. When a connection is established between **tvdsvr** and TotalView, the 64-bit password passed by TotalView must match the password set with this option. When the password is set, **tvdsvr** displays the selected number in the following message:

```
pw = hexnumhigh:hexnumlow
```

We recommend using this option to avoid connections by other users.

**NOTE** If necessary, you can disable password checking by specifying the “**—set\_pw 0:0**” option with the **tvdsvr** command. Disabling password checking is dangerous; it

—working\_directory *directory*

allows anyone to connect to your server and start programs, including shell commands, using your UID. Therefore, we do not recommend disabling password checking.

—set\_pws *password-list*

Sets 64-bit passwords. TotalView must supply these passwords when **tvdsvr** establishes the connection with it. The argument to this command is a comma-separated list of passwords that TotalView automatically generates. This option is most often used with a bulk launch.

—verbosity *level*

Sets the verbosity level of TotalView Debugger Server-generated messages to *level*, which may be one of **silent**, **error**, **warning**, or **info**.

Default: **info**

—working\_directory *directory*

Makes *directory* the directory to which TotalView will be connected.

Note that the command assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on both the host and target machines.

After performing this operation, the TotalView Debugger Server is started.

## Replacement Characters

When placing a **tvdsvr** command within a **Server Launch** or **Bulk Launch** string (see **File > Preferences** for more information), you will need to use special replacement characters. When your program needs to launch a remote process, TotalView replaces these command characters with what they represent. Here are the replacement characters:

.....  
%C

%C	Is replaced by the name of the server launch command being used. On most platforms, this is <b>rsh</b> . On HP, this command is <b>remsh</b> . If the <b>TVDSVRLAUNCHCMD</b> environment variable exists, TotalView will use its value instead of its platform-specific value.
%D	Is replaced by the absolute path name of the directory to which TotalView will be connected.
%H	Expands to the host name of the machine upon which TotalView is running. (This replacement character is most often used in bulk server launch commands. However, it can be used in a regular server launch and within a <b>tvdsvr</b> command contained within a temporary file.)
%L	<p>If TotalView is launching one process, this is replaced by the host name and TCP/IP port number (<i>host-name:port</i>) on which TotalView is listening for connections from <b>tvdsvr</b>.</p> <p>If a bulk launch is being performed, TotalView replaces this with a comma-separated list of the host names and TCP/IP port numbers (<i>hostname:port,host-name:port...</i>) on which TotalView is listening for connections from <b>tvdsvr</b>.</p>
%N	Is replaced by the number of servers that will be launched. This is only used in a bulk server launch command.
%P	<p>If TotalView is launching one process, this is replaced by the password that TotalView automatically generated.</p> <p>If a bulk launch is being performed, TotalView replaces this with a comma-separated list of 64-bit passwords.</p>
%R	Is replaced by the host name of the remote machine that was specified in the <b>File &gt; New Program</b> command.
%S	If TotalView is launching one process, this is replaced by the port number on the machine upon which the debugger is running.

If a bulk server launch is being performed, TotalView replaces this with a comma-separated list of port numbers.

**%t1 and %t2**

Is replaced by files that TotalView creates containing information it generates. This is only available in a bulk launch.

These temporary files have the following structure:

- (1) An optional header line containing initialization commands required by your system.
- (2) One line for each host being connected to, containing host-specific information.
- (3) An optional trailer line containing information needed by your system to terminate the temporary file.

The **File > Preferences Bulk Server** Page allows you to define templates for the actions performed by temporary files. These files will use these replacement characters. You can only use the **%N**, **%t1**, and **%t2** replacement characters within header and trailer lines of temporary files. The **%L**, **%P**, and **%S** characters can be used in header or trailer lines or within a host line defining the command that initiates a single-process server launch.

The templates for temporary files can also be set using X resources.

**%V**

Is replaced by the current TotalView verbosity setting.

.....  
%V

# Compilers and Platforms

This appendix describes the compilers and parallel runtime environments used on platforms supported by TotalView. You must refer to the TotalView release notes included in the TotalView distribution for information on the specific compiler and runtime environment supported by TotalView.

For information on supported operating systems, please refer to Appendix B *"Operating Systems"* on page 321.

Topics in this appendix are:

- Compiling with Debugging Symbols
- Using Exception Data on Compaq Tru64 UNIX
- Linking with the dbfork Library

## Compiling with Debugging Symbols

You need to compile programs with the `-g` option and possibly other compiler options so that debugging symbols are included. This section shows the specific compiler commands to use for each compiler that TotalView supports.

**NOTE** Please refer to the release notes in your TotalView distribution for the latest information about supported versions of the compilers and parallel runtime environments listed here.

## Compaq Tru64 UNIX

Table 23 lists the procedures to compile programs on Compaq Tru64 UNIX.

TABLE 23: Compiling with Debugging Symbols on Compaq Tru64 UNIX

Compiler	Compiler Command Line
Compaq Tru64 UNIX C	<code>cc -g -c program.c</code>
Compaq Tru64 UNIX C++	<code>cxx -g -c program.cxx</code>
Compaq Tru64 UNIX Fortran 77	<code>f77 -g -c program.f</code>
Compaq Tru64 UNIX Fortran 90	<code>f90 -g -c program.f90</code>
GCC EGCS C	<code>gcc -g -c program.c</code>
GCC EGCS C++	<code>g++ -g -c program.cxx</code>
KAI C	<code>KCC +K0 -c program.c</code>
KAI C++	<code>KCC +K0 -c program.cxx</code>
KAI Guide C (OpenMP)	<code>guidec -g +K0 program.c</code>
KAI Guide C++ (OpenMP)	<code>guidec -g +K0 program.cxx</code>
KAI Guide F77 (OpenMP)	<code>guidef77 -g -WG,-cmpo=i program.f</code>

When compiling with KCC for debugging, we recommend that you use the `+K0` option and not the `-g` option. Also, the `-WG,-cmpo=i` option to the `guidef77` command may not be required on all versions because `-g` can imply these options.

## HP-UX

Table 24 lists the procedures to compile programs on HP-UX.

TABLE 24: Compiling with Debugging Symbols on HP-UX

Compiler	Compiler Command Line
HP ANSI C	<code>cc -g -c program.c</code>
HP C++	<code>aCC -g -c program.cxx</code>
HP Fortran 90	<code>f90 -g -c program.f90</code>
KAI C	<code>KCC +K0 -c program.c</code>
KAI C++	<code>KCC +K0 -c program.cxx</code>
KAI Guide C (OpenMP)	<code>guidec -g +K0 program.c</code>



TABLE 24: Compiling with Debugging Symbols on HP-UX (cont.)

Compiler	Compiler Command Line
KAI Guide C++ (OpenMP)	<b>guidec -g +K0</b> <i>program.cxx</i>
KAI Guide F77 (OpenMP)	<b>guidef77 -g -WG,-cmpo=i</b> <i>program.f</i>

When compiling with KCC for debugging, we recommend that you use the **+K0** option and not the **-g** option. Also, the **-WG,-cmpo=i** option to the **guidef77** command may not be required on all versions because **-g** can imply these options.

## IBM AIX on RS/6000 Systems

Table 25 lists the procedures to compile programs on IBM RS/6000 systems running AIX.

TABLE 25: Compiling with Debugging Symbols on AIX

Compiler	Compiler Command Line
GCC EGCS C	<b>gcc -g -c</b> <i>program.c</i>
GCC EGCS C++	<b>g++ -g -c</b> <i>program.cxx</i>
IBM xlc C	<b>xlc -g -c</b> <i>program.c</i>
IBM xlc C++	<b>xlc -g -c</b> <i>program.cxx</i>
IBM xlf Fortran 77	<b>xlf -g -c</b> <i>program.f</i>
IBM xlf90 Fortran 90	<b>xlf90 -g -c</b> <i>program.f90</i>
KAI C	<b>KCC +K0 -qnofullpath -c</b> <i>program.c</i>
KAI C++	<b>KCC +K0 -qnofullpath -c</b> <i>program.cxx</i>
KAI Guide C (OpenMP)	<b>guidec -g +K0</b> <i>program.c</i>
KAI Guide C++ (OpenMP)	<b>guidec -g +K0</b> <i>program.cxx</i>
KAI Guide F77 (OpenMP)	<b>guidef77 -g -WG,-cmpo=i</b> <i>program.f</i>
Portland Group HPF	<b>pghpf -g -Mtv -c</b> <i>program.hpf</i>

If TotalView supports threading, you should not define any of the following variables:

- AIXTHREAD\_DEBUG
- AIXTHREAD\_COND\_DEBUG

- AIXTHREAD\_MUTEX\_DEBUG
- AIXTHREAD\_RWLOCK\_DEBUG

When compiling with KCC, you must specify the **-qnofullpath** option; KCC is a preprocessor that passes its output to the IBM xlc C compiler. It will discard **#line** directives necessary for source-level debugging if **-qfullpath** is specified. We also recommend that you use the **+K0** option and not the **-g** option.

When compiling with **guidef77**, the **-WG,-cmpto=i** option may not be required on all versions because **-g** can imply these options.

When compiling Fortran programs with the C preprocessor, pass the **-d** option to the compiler driver. For example: **xlf -d -g -c program.F**

When compiling with any of the IBM xl compilers, if your program will be moved from its creation directory, or you do not want to set the search directory path during debugging, use the **-qfullpath** compiler option. For example:

```
xlf -qfullpath -g -c program.f
```

## SGI IRIX-MIPS Systems

Table 26 lists the procedures to compile programs on SGI MIPS systems running IRIX.

TABLE 26: Compiling with Debugging Symbols on IRIX-MIPS

Compiler	Compiler Command Line
GCC EGCS C	<b>gcc -g -c <i>program.c</i></b>
GCC EGCS C++	<b>gcc -g -c <i>program.cxx</i></b>
KAI C	<b>KCC +K0 -c <i>program.c</i></b>
KAI C++	<b>KCC +K0 -c <i>program.cxx</i></b>
KAI Guide C (OpenMP)	<b>guidec -g +K0 <i>program.c</i></b>
KAI Guide C++ (OpenMP)	<b>guidec -g +K0 <i>program.cxx</i></b>
KAI Guide F77 (OpenMP)	<b>guidef77 -g -WG,-cmpto=i <i>program.f</i></b>
Portland Group HPF	<b>pghpf -g -64 -Mtv -c <i>program.hpf</i></b>
SGI MIPSpro 90	<b>f90 -n32 -g -c <i>program.f90</i></b> <b>f90 -64 -g -c <i>program.f90</i></b>

TABLE 26: Compiling with Debugging Symbols on IRIX-MIPS (cont.)

Compiler	Compiler Command Line
SGI MIPSpro C	<code>cc -n32 -g -c program.c</code> <code>cc -64 -g -c program.c</code>
SGI MIPSpro C++	<code>CC -n32 -g -c program.cxx</code> <code>CC -64 -g -c program.cxx</code>
SGI MIPSpro77	<code>f77 -n32 -g -c program.f</code> <code>f77 -64 -g -c program.f</code>

Compiling with `-n32` or `-64` is supported. TotalView does not support compiling with `-32`, which is the default for some compilers. You must specify either `-n32` or `-64`.

When compiling with KCC for debugging, we recommend that you use the `+K0` option and not the `-g` option. Also, the `-WG,-cmpo=i` option to the `guidef77` command may not be required on all versions because `-g` can imply these options.

You must compile your programs with the `pghpf -64` compiler option; on SGI IRIX, TotalView can debug 64-bit executables only.

## SunOS 5 on SPARC

Table 27 lists the procedures to compile programs on SunOS 5 SPARC.

TABLE 27: Compiling with Debugging Symbols on SunOS 5

Compiler	Compiler Command Line
Apogee C	<code>apcc -g -c program.c</code>
Apogee C++	<code>apcc -g -c program.cxx</code>
GCC EGCS C	<code>gcc -g -c program.c</code>
GCC EGCS C++	<code>g++ -g -c program.cxx</code>
KAI C	<code>KCC +K0 -c program.c</code>
KAI C++	<code>KCC +K0 -c program.cxx</code>
KAI Guide C (OpenMP)	<code>guidec -g +K0 program.c</code>
KAI Guide C++ (OpenMP)	<code>guidec -g +K0 program.cxx</code>
KAI Guide F77 (OpenMP)	<code>guidef77 -g -WG,-cmpo=i program.f</code>
Portland Group HPF	<code>pghpf -g -Mtv -c program.hpf</code>

TABLE 27: Compiling with Debugging Symbols on SunOS 5 (cont.)

Compiler	Compiler Command Line
SunPro/WorkShop C	<code>cc -g -c program.c</code>
SunPro/WorkShop C++	<code>CC -g -c program.cxx</code>
SunPro/WorkShop Fortran 77	<code>f77 -g -c program.f</code>
WorkShop Fortran 90	<code>f90 -g -c program.f90</code>

When compiling with KCC for debugging, we recommend that you use the `+K0` option and not the `-g` option. Also, the `-WG,-cmpto=i` option to the `guidf77` command may not be required on all versions because `-g` can imply these options.

## Using Exception Data on Compaq Tru64 UNIX

If you receive the following error message when you load an executable into TotalView, you may need to compile your program so that exception data is included:

Cannot find exception information. Stack backtraces may not be correct.

To provide a complete stack backtrace in all situations, TotalView needs the exception data to be included in the compiled executable. To compile with exception data, you need to use the following options:

```
cc -WI,-u,_fpdata_size program.c
```

where:

- `-WI` Passes the arguments that follow to another compilation phase (`-W`), which in this case is the linker (`l`). Each argument is separated by a comma (`,`).
- `-u` Causes the linker to mark the next argument (`_fpdata_size`) as undefined.
- `_fpdata_size` Marks the `_fpdata_size` variable as undefined, which forces the exception data into the executable.
- `program.c` Is the name of your program.

Compiling with exception data increases the size of your executable slightly. If you choose not to compile with exception data, TotalView can provide correct stack backtraces in most situations, but not in all situations.

## Linking with the dbfork Library

If your program uses the **fork()** and **execve()** system calls, and you want to debug the child processes, you need to link programs with the **dbfork** library.

### Compaq Tru64 UNIX

Add one of the following arguments to the command that you use to link your programs:

- `/opt/totalview/lib/libdbfork.a`
- `-L/opt/totalview/lib -ldbfork`

For example:

```
cc -o program program.c -L/opt/totalview/lib -ldbfork
```

As an alternative, you can set the `LD_LIBRARY_PATH` environment variable and omit the `-L` option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

### HP-UX

Add either the `-ldbfork` or `-ldbfork_64` argument to the command that you use to link your programs. If you are compiling 32-bit code, use one of the following arguments:

- `/opt/totalview/lib/libdbfork.a`
- `-L/opt/totalview/lib -ldbfork`

For example:

```
cc -n32 -o program program.c -L/opt/totalview/lib -ldbfork
```

If you are compiling 64-bit code, use the following arguments:

- `/opt/totalview/lib/libdbfork_64.a`
- `-L/opt/totalview/lib -ldbfork_64`

For example:

```
cc -64 -o program program.c -L/opt/totalview/lib -ldbfork_64
```

As an alternative, you can set the `LD_LIBRARY_PATH` environment variable and omit the `-L` command-line option. For example:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

## IBM AIX on RS/6000 Systems

Add either the `-dbfork` or `-ldbfork_64` argument to the command that you use to link your programs. If you are compiling 32-bit code, use the following arguments:

- `/usr/totalview/lib/libdbfork.a -bkeepfile:/usr/totalview/lib/libdbfork.a`
- `-L/usr/totalview/lib -ldbfork -bkeepfile:/usr/totalview/lib/libdbfork.a`

For example:

```
cc -o program program.c \
    -L/usr/totalview/lib -ldbfork \
    -bkeepfile:/usr/totalview/lib/libdbfork.a
```

If you are compiling 64-bit code, use the following arguments:

- `/usr/totalview/lib/libdbfork_64.a \`  
`-bkeepfile:/usr/totalview/lib/libdbfork.a`
- `-L/usr/totalview/lib -ldbfork_64 \`  
`-bkeepfile:/usr/totalview/lib/libdbfork.a`

For example:

```
cc -o program program.c \
    -L/usr/totalview/lib -ldbfork \
    -bkeepfile:/usr/totalview/lib/libdbfork.a
```

When you use **gcc** or **g++**, use the **-Wl,-bkeepfile** option instead of using the **-bkeepfile** option, which will pass the same option to the binder. For example:

```
gcc -o program program.c -L/usr/totalview/lib -ldbfork \
    -Wl,-bkeepfile:/usr/totalview/lib/libdbfork.a
```

## Linking C++ Programs with dbfork

The binder option **-bkeepfile** currently cannot be used with the IBM xLC C++ compiler. The compiler passes all binder options to an additional pass called **munch**, which cannot handle the **-bkeepfile** option.

To work around this problem, we have provided the C++ header file **libdbfork.h**. You must include this file somewhere in your C++ program, in order to force the components of the **dbfork** library to be kept in your executable. The file **libdbfork.h** is included only with the RS/6000 version of TotalView. This means that if you are creating a program that will run on more than one platform, you should place the **include** within an **#ifdef** statement. For example:

```
#ifdef _AIX
#include "/usr/totalview/lib/libdbfork.h"
#endif
int main (int argc, char *argv[])
{
}
```

In this case, you would not use the **-bkeepfile** option and would instead link your program using one of the following options:

- **/usr/totalview/lib/libdbfork.a**
- **-L/usr/totalview/lib -ldbfork**

## SGI IRIX6-MIPS

Add one of the following arguments to the command that you use to link your programs.

If you are compiling your code with **-n32**, use the following arguments:

- **/opt/totalview/lib/libdbfork\_n32.a**
- **-L/opt/totalview/lib -ldbfork\_n32**

For example:

```
cc -n32 -o program program.c -L/opt/totalview/lib -ldbfork_n32
```

If you are compiling your code with **-64**, use the following arguments:

- **/opt/totalview/lib/libdbfork.a\_n64.a**
- **-L/opt/totalview/lib -ldbfork\_n64**

For example:

```
cc -64 -o program program.c -L/opt/totalview/lib -ldbfork_n64
```

As an alternative, you can set the **LD\_LIBRARY\_PATH** environment variable and omit the **-L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```

## SunOS 5 SPARC

Add one of the following arguments to the command that you use to link your programs:

- **/opt/totalview/lib/libdbfork.a**
- **-L/opt/totalview/lib -ldbfork**

For example:

```
cc -o program program.c -L/opt/totalview/lib -ldbfork
```

As an alternative, you can set the **LD\_LIBRARY\_PATH** environment variable and omit the **-L** option on the command line:

```
setenv LD_LIBRARY_PATH /opt/totalview/lib
```



# Operating Systems

This appendix describes the operating system features that can be used with TotalView. This appendix includes the following topics:

- Supported Operating Systems
- Mounting the /proc File System (Compaq Tru64 UNIX, IRIX, and SunOS 5 only)
- Swap Space
- Shared Libraries
- Debugging Dynamically Loaded Libraries
- Remapping Keys (Sun Keyboards only)
- Expression System

## Supported Operating Systems

Here is an overview of operating systems and some of the environments supported by TotalView at the time when this book was printed. You should know that this list changes frequently. For a complete list of hardware and software requirements, see the TOTALVIEW PLATFORMS document in your software distribution.

- Compaq Alpha workstations running Compaq Tru64 UNIX versions V4.0D, V4.0E, V4.0F, V5.0, V5.0A, and V5.1. All versions require patches. See "Compaq UNIX Patch Procedures" in the TOTALVIEW PLATFORMS document for instructions.
- HP PA-RISC 1.1 or 2.0 systems running HP-UX Version 11.00 and 11.10.
- IBM RS/6000 and SP systems running AIX versions 4.2.1, 4.3, 4.3.1, 4.3.2, and 4.3.3.

- Linux Red Hat 6.0, 6.1, 6.2, 7.0, and 7.1.
- SGI IRIX 6.2, 6.3, 6.4, or 6.5 on any MIPS R4000, R4400, R4600, R5000, R8000, R10000, and R12000 processor-based systems.
- Sun Sparc SunOS 5 (Solaris 2.x) systems running SunOS versions 5.5, 5.5.1, 5.6, 5.7, and 5.8. (Solaris 2.5, 2.5.1, or 2.6, 7, or 8). TotalView also supports QSW CS-2 based on Sparc Solaris 2.5.1 or 2.6.

**NOTE** TotalView on QSW CS-2 is nearly identical to TotalView on Sun Solaris 2.x systems.

## Mounting the /proc File System

To debug programs on Compaq Tru64 UNIX, SunOS 5, and IRIX with TotalView, you need to mount the **/proc** file system.

If you receive one of the following errors from TotalView, the **/proc** file system might not be mounted:

- job\_t::launch, creating process: process not found
- Error launching process while trying to read dynamic symbols
- Creating Process... Process not found  
Clearing Thrown Flag  
Operation Attempted on an unbound d\_process object

To determine whether the **/proc** file system is mounted, enter the appropriate command from the following table.

**Table 28: Commands for Determining Whether /proc Is Mounted**

Operating System	Command
Compaq Tru64 UNIX	% /sbin/mount -t procfs /proc on /proc type procfs (rw)
SunOS 5	% /sbin/mount   grep /proc /proc on /proc read/write/setuid on ...
IRIX	% /sbin/mount   grep /proc /proc on /proc type proc (rw)

If you receive one of these messages from the **mount** command, the **/proc** file system is mounted.

## Compaq Tru64 UNIX and SunOS 5

To make sure that the **/proc** file system is mounted each time your system boots, add the appropriate line from the following table to the appropriate file.

Table 29: Commands for Automatically Mounting the **/proc** File System

Operating System	Name of File	Line to add
Compaq Tru64 UNIX	/etc/fstab	/proc /proc procfs rw 0 0
SunOS 5	/etc/vfstab	/proc - /proc proc - no -

Then, to mount the **/proc** file system, enter the following command:

```
/sbin/mount /proc
```

## SGI IRIX

To make sure that the **/proc** file system is mounted each time your system boots, make sure that **/etc/rc2** issues the **/etc/mntproc** command. Then, to mount the **/proc** file system, enter the following command:

```
/etc/mntproc
```

## Swap Space

Debugging large programs can exhaust the swap space on your machine. If you run out of swap space, TotalView exits with a fatal error, such as:

### ■ Fatal Error: Out of space trying to allocate

This error indicates that TotalView failed to allocate dynamic memory. It can occur anytime during a TotalView session. It can also indicate that the data size limit in the C shell is too small. You can use the C shell's **limit** command to increase the data size limit. For example:

```
limit datasize unlimited
```

### ■ job\_t::launch, creating process: Operation failed

This error indicates that the **fork()** or **execve()** system call failed while TotalView was creating a process to debug. It can happen when TotalView tries to create a process.

## Compaq Tru64 UNIX

To find out how much swap space has been allocated and is currently being used, use the **swapon** command on Compaq Tru64 UNIX:

```
% /sbin/swapon -s
```

Total swap allocation:

Allocated space: 85170 pages (665MB)

Reserved space: 14216 pages ( 16%)

Available space: 70954 pages ( 83%)

Swap partition /dev/rz3b:

Allocated space: 16384 pages (128MB)

In-use space: 2610 pages ( 15%)

Free space: 13774 pages ( 84%)

Swap partition /dev/rz3h:

Allocated space: 52402 pages (409MB)

In-use space: 2575 pages ( 4%)

Free space: 49827 pages ( 95%)

Swap partition /dev/rz1b:

Allocated space: 16384 pages (128MB)

In-use space: 2592 pages ( 15%)

Free space: 13792 pages ( 84%)

In this example, 665 MB of swap space is allocated, and 106 MB of it is currently in use.

To find out how much swap space is in use while you are running TotalView:

```
/bin/ps -o LFMT
```

For example, in this case the value in the VSZ column is 4.45 MB:

```
UID PID PPID CP PRI NI VSZ RSS ...
12270 5340 5293 0 41 0 4.45M 1.27 ...
```

To add swap space, use the **/sbin/swapon(8)** command. You must be **root** to use this command. For more information, refer to the online manual page for this command.

## HP HP-UX

The **swapinfo** command on an HP-UX system lets you find out how much swap space is allocated and is being used. For example:

```
# /usr/sbin/swapinfo
          Kb   Kb   Kb  PCT  START/   Kb
TYPE    AVAIL  USED  FREE  USED  LIMIT RESERVE PRI NAME
dev    1048576    0 1048576   0%    0    -    1 /dev/vg00/lvol2
reserve    - 389240 -389240
memory 1178960 966564 212396  82%
```

To find out how much swap space is being used while TotalView is running, enter:

```
/usr/bin/ps -lf
```

Here is an example of what you might see:

```
  F S   UID   PID   PPID  C PRI  NI   ADDR   SZ ...
21 T    rtf  4414  13709  0 154  20  ce8d800 2764 ...
```

The **SZ** column shows the pages occupied by a program.

To add swap space, use the **/usr/sbin/swapon(1M)** command or the **SAM** (System Administration Manager) utility. If you use **SAM**, invoke the **Swap** command within the **Disks and File Systems** menu.

## Maximum Data Size

To see the current data size limit in the C shell, enter:

```
limit datasize
```

The following command displays the current *hard* limit:

```
limit -h datasize
```

If the current limit is lower than the hard limit, you can easily raise the current limit. To change the current limit, enter:

```
limit datasize new_data_size
```

If the hard limit is too low, you must reconfigure and rebuild the kernel, and then reboot. This is most easily done using **SAM**.

To change **maxdsiz**, use the following path through the **SAM** menus:

Kernel Configuration > Configurable Parameters > maxdsiz >  
 Actions > Modify Configurable Parameter >  
 Specify New Formula/Value > Formula/Value

You can now enter the new maximum data segment size.

You may also need to change the value for **maxdsiz\_64**.

Here is the command that lets you rebuild the kernel with these changed values:

Configurable Parameter > Actions > Process New Kernel

Answer **yes** to process the kernel modifications, **yes** to install the new kernel, and **yes** again to reboot the machine with the new kernel.

When the machine reboots, the value you set for **maxdsiz** should be the new hard limit.

## IBM AIX

To find out how much swap space has been allocated and is currently being used, use the **pstat** command:

```
% /usr/sbin/pstat -s
```

PAGE SPACE:

USED PAGES	FREE PAGES
7555	115325

In this example, 122880 (7555 + 115325) pages of swap space was allocated; 7555 pages are currently in use, and 115325 pages are free.

To find out how much swap space is in use while you are running TotalView:

- 1 Start TotalView with a large executable:  

```
totalview executable
```
- 2 Press Ctrl-Z to suspend TotalView.
- 3 Use the following command to see how much swap space TotalView is using:  

```
ps u
```

For example, in this case the value in the SZ column is 5476 KB:

```
USER  PID %CPU %MEM  SZ  RSS  TTY ...
smith 15080 0.0 6.0 5476 5476 pts/1 ...
```

To add swap space, use the AIX system management tool, **smit**. Use the following path through the **smit** menus:

System Storage Management → Logical Volume Manager →  
Paging Space

## Linux

To find out how much swap space has been allocated and is currently being used, use either the **swapon** or **top** commands on Linux:

```
% /sbin/swapon -s
```

```
Filename      Type      Size    Used    Priority
/dev/hda7     partition 128484  28      -1
```

```
% top
```

```
jcownie@pc2: top
```

```
(null) 1:29pm up 4:28, 1 user, load average: 0.00, 0.00, 0.00
```

```
52 processes: 50 sleeping, 2 running, 0 zombie, 0 stopped
```

```
CPU states: 1.1% user, 0.4% system, 0.0% nice, 98.4% idle
```

```
Mem: 127904K a, 116512K used, 11392K free, 36020K shrd, \
      3632K buff
```

```
Swap: 128484K av, 28K used, 128456K free 79804K
cached
```

```
... remainder of "top" listing removed ...
```

You can use the **mkswap**(8) command to create swap space. The **swapon**(8) command tells Linux that it should use this space.

## SGI IRIX

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
% /sbin/swap -s
```

```
total: 1.55m allocated + 124.47m add'l reserved = 126.02m bytes
used, 250.94m bytes available
```

To find out how much swap space is in use while you are running TotalView:

- 1 Start TotalView with a large executable:

**totalview** *executable*

- 2 Press Ctrl-Z to suspend TotalView.

- 3 Use the following command to see how much swap space TotalView is using:

`/bin/ps -l`

For example, in this case the value in the SZ column is 584 pages.

```
F S  UID  PID  PPID  C  PRI  NI  P  ...
b0 T 14694 26236 26271  5   62  20  *  ...
```

Use the following command to determine the number of bytes in a page:

`sysconf PAGESIZE`

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be root to use these commands. For more information, refer to the on-line manual pages for these commands.

## SunOS 5

To find out how much swap space has been allocated and is currently being used, use the **swap** command:

```
% /usr/sbin/swap -s
total: 16192K bytes allocated + 7140K bytes \
      reserved = 23332K used, 63456K available
```

To find out how much swap space is in use while you are running TotalView:

- 1 Start TotalView with a large executable:

**totalview** *executable*

- 2 Press Ctrl-Z to suspend TotalView.

- 3 Use the following command to see how much swap space TotalView is using:

`/bin/ps -l`

To add swap space, use the **mkfile(1M)** and **swap(1M)** commands. You must be **root** to use these commands. For more information, refer to the on-line manual pages for these commands.



## Shared Libraries

TotalView supports dynamically linked executables, that is, executables that are linked with shared libraries.

When you start TotalView with a dynamically linked executable, TotalView loads an additional set of symbols for the shared libraries, as indicated in the shell from which you started TotalView. To accomplish this, TotalView:

- 1 Runs a sample process and discards it.
- 2 Reads information from the process.
- 3 Reads the symbol table for each library.

When you create a process without starting it, and the process does not include shared libraries, the PC points to the entry point of the process, usually the **start** routine. If the process does include shared libraries, however, TotalView takes the following actions:

- Runs the dynamic loader (SunOS 5: **ld.so**, Compaq Tru64 UNIX: **/sbin/loader**, Linux: **/lib/ld-linux.so.?**, IRIX: **rld**).
- Sets the PC to point to the location after the invocation of the dynamic loader but before the invocation of C++ static constructors or the **main** routine.

**NOTE** ON HP-UX, TotalView cannot stop the loading of shared libraries until after static constructors on shared library initialization routines have been run.

When you attach to a process that uses shared libraries, TotalView takes the following actions:

- If you attached to the process after the dynamic loader ran, then TotalView loads the dynamic symbols for the shared library.
- If you attached to the process before it runs the dynamic loader, TotalView allows the process to run the dynamic loader to completion. Then, TotalView loads the dynamic symbols for the shared library.

If desired, you can suppress the use of shared libraries by starting TotalView with the **-no\_dynamic** option. Refer to Chapter 13 "TotalView Command Syntax" on page 289 for details on this TotalView startup option.

If you believe that a shared library has changed since you started a TotalView session, you can use the **Group > Rescan Library** command to reload library symbol tables. Be aware that only some systems such as AIX permit you to reload library information.

## Changing Linkage Table Entries and LD\_BIND\_NOW

If you are executing a dynamically linked program, calls from the executable into a shared library are made using the *Procedure Linkage Table* (PLT). Each function in the dynamic library that is called by the main program has an entry in this table. Normally, the dynamic linker fills the PLT entries with code that calls the dynamic linker. This means that the first time that your code calls a function in a dynamic library, the runtime environment calls the dynamic linker. The linker will then modify the entry so that next time this function is called, it will not be involved.

This is not the behavior you want or expect when debugging a program because TotalView will do one of the following:

- Place you within the dynamic linker (which you don't want to see).
- Step over the function.

And, because the entry is altered, everything appears to work fine the next time you step into this function.

On most operating systems (except HP), you can correct this problem by setting the **LD\_BIND\_NOW** environment variable. For example:

```
setenv LD_BIND_NOW 1
```

This tells the dynamic linker that it should alter the PLT when the program starts executing rather than doing it when the program calls the function.

HP-UX does not have this (or an equivalent) variable. On HP systems, you can avoid this problem by linking the executable being debugged with the **-B immediate** option or by invoking **chatr** with the **-B immediate** option. (See the **chatr** documentation for complete information on how to use this command.)

You will also have to enter **pxdb -s** on.

## Using Shared Libraries on HP-UX

The dynamic library loader on HP-UX loads shared libraries into shared memory. Writing breakpoints into code sections loaded in shared memory can cause programs not under TotalView's control to fail when they execute an unexpected breakpoint.

If you need to single-step or set breakpoints in shared libraries, you must set your application to load those libraries in private memory. This is done using HP's **pxdb** command.

```
pxdb -s on appname (load shared libraries into private memory)
pxdb -s off appname (load shared libraries into shared memory)
```

For 64-bit platforms, use **pxdb64** instead of **pxdb**. If the version of **pxdb64** supplied with HP's compilers does not work correctly, you may need to install an HP-supplied patch. You will find additional information on the TOTALVIEW RELEASE NOTES.

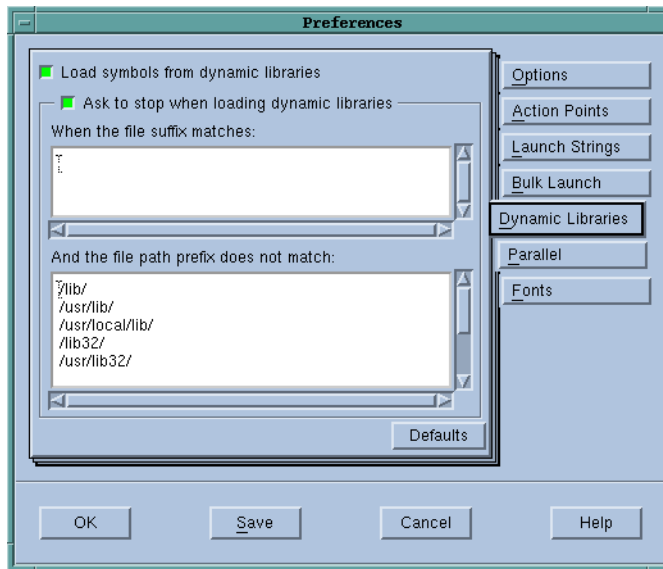
## Debugging Dynamically Loaded Libraries

TotalView automatically reads the symbols of shared libraries that are dynamically loaded into your program at runtime. These libraries are ones that are loaded using **dlopen** (or, on IBM AIX, **load** and **loadbind**).

TotalView automatically detects these calls, and then loads the symbol table from the newly loaded libraries and plants any enabled saved breakpoints for these libraries. TotalView then decides whether to ask you about stopping the process to plant breakpoints. You will set these characteristics by using the **Dynamic Libraries** Page within the **File > Preferences** dialog box. (See "*File > Preferences Dialog Box: Dynamic Libraries Page*" on page 332.)

TotalView decides according to the following rules:

- 1 If the "**Load symbols from dynamic libraries** preference is set to **false**, TotalView *does not* ask you about stopping.



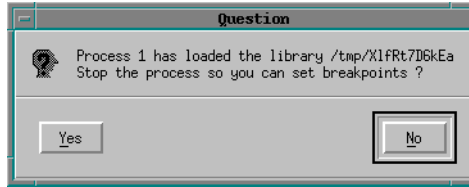
**Figure 136: File > Preferences Dialog Box: Dynamic Libraries Page**

- 2 If one or more of the strings in the “When the file suffix matches” preference list is a suffix of the full library name (including path), TotalView asks you about stopping.
- 3 If one or more of the strings in the “When the file path prefix does not match” list is a prefix of the full library name (including path), TotalView does not ask you about stopping.
- 4 If the newly loaded libraries have any saved breakpoints, TotalView does not ask you about stopping.
- 5 If none of the rules above apply, TotalView asks you about stopping.

If TotalView does not ask you about stopping the process, the process is continued.

If TotalView decides to ask you about stopping, it displays a dialog box, asking if it should stop the process so you can set breakpoints. To stop the process, answer **yes**.

To allow the process to continue executing, answer **no**. Stopping the process allows you to insert breakpoints in the newly loaded shared library.



**Figure 137: Stop Process Question**

You can tell TotalView if it should ask by doing either or both of the following:

- You can set the `-ask_on_dlopen` command-line option to **true**, or you can set the `-no_ask_on_dlopen` option to false.
- Set the **Load Symbols from dynamic libraries** preference.

The following table lists paths where you are not asked if it is alright to load dynamic link libraries:

**Table 30: Default “Don’t Ask” on Load List**

Platform	Value	
Compaq Tru64 UNIX Alpha	/usr/shlib/	/usr/ccs/lib/
	/usr/lib/cmplrs/cc/	/usr/lib/
HP-UX	/usr/local/lib/	/var/shlib/
	/usr/lib/	/usr/lib/pa20_64
	/opt/langtools/lib/	/opt/langtools/lib/pa20_64/
IBM AIX	/lib/	/usr/lib/
	/usr/lpp/	/usr/ccs/lib/
	/usr/dt/lib/	/tmp/
SGI IRIX	/lib/	/usr/lib/
	/usr/local/lib/	/lib32/
	/usr/lib32/	/usr/local/lib32/
	/lib64/	/usr/lib64/
SUN Solaris 2.x	/usr/local/lib64	
	/lib/	/usr/lib/
Linux x86	/usr/ccs/lib/	
	/lib	/usr/lib
Linux Alpha	/lib	/usr/lib

The values you enter on the TotalView preference should be space-separated lists of the prefixes and suffixes to be used.

After starting TotalView, you can change these lists by using the **When the file suffix matches** and **And the file path prefix does not match** preferences.

## Known Limitations

Dynamic library support has the following known limitations:

- TotalView does not deal correctly with parallel programs that call **dlopen** on different libraries in different processes. TotalView requires that the processes have a uniform address space, including all shared libraries.
- TotalView does not yet fully support unloading libraries (using **dlclose**) and then reloading them at a different address using **dlopen**.

## Remapping Keys

On the SunOS 5 keyboard, you may need to remap the page-up and page-down keys to the prior and next keysym so that you can scroll TotalView windows with the page-up and page-down keys. To do so, add the following lines to your X Window System start-up file:

```
# Remap F29/F35 to PgUp/PgDn
xmodmap -e 'keysym F29 = Prior'
xmodmap -e 'keysym F35 = Next'
```

## Expression System

Depending on the target platform, TotalView supports:

- An interpreted expression system only
- Both an interpreted and a compiled expression system

Unless stated otherwise below, TotalView supports interpreted expressions only.

## Compaq Alpha Tru64 UNIX

On Compaq Tru64 UNIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

## IBM AIX

On IBM AIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

Some program functions called from the TotalView expression system on the Power architecture cannot have floating-point arguments that are passed by value. However, in functions with a variable number of arguments, floating-point arguments *can* be in the varying part of the argument list. For example, you can include floating-point arguments with calls to `printf`:

```
double d = 3.14159;
printf("d = %f\n", d);
```

## SGI IRIX

On IRIX, TotalView supports compiled and interpreted expressions. TotalView also supports assembler in expressions.

TotalView includes the SGI IRIX expression compiler. This feature does not use any MIPS-IV specific instructions. It does use MIPS-III instructions freely. It fully supports **-n32** and **-64** executables.

Due to limitations in dynamically allocating patch space, compiled expressions are disabled by default on SGI IRIX. To enable compiled expressions in an invocation of TotalView, use the X resource **totalview\*compileExpressions** on page 277 to set the option to **true**, or pass the X resource as the **-compileExpressions=true** command-line option. This option also tells TotalView to find or allocate patch space in your program for code fragments generated by the expression compiler.

If you enable compiled patches on SGI IRIX with a multiprocess program, you must use static patches. For example, if you link a static patch space into an IRIX MPI program and run the program under TotalView's control,

TotalView should let you debug it. If you attach to a previously started MPI job, however, even static patches will not let the program run properly. If TotalView still fails to work properly with the static patch space, then you probably cannot use compiled patches with your program.

For general instructions on using patch space allocation controls with compiled expressions, see *"Allocating Patch Space for Compiled Expressions"* on page 221.



# Architectures

This appendix describes the architectures TotalView supports, including:

- Compaq Alpha
- HP PA-RISC
- IBM Power
- Intel-x86 (Intel 80386, 80486 and Pentium processors)
- SGI MIPS
- Sun SPARC

It includes the following topics for each architecture:

- General registers
- Floating-point registers
- Floating-point format

## Compaq Alpha

This section contains the following information:

- Alpha General Registers
- Alpha Floating-Point Registers
- Alpha FPCR Register

**NOTE** The Alpha processor supports the IEEE floating-point format.

## Alpha General Registers

TotalView displays the Alpha general registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each general register, and the actions you can take with each register.

**Table 31: Alpha General Purpose Integer Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
V0	Function value register	<long>	yes	yes	\$v0
T0 – T7	Conventional scratch registers	<long>	yes	yes	\$t0 – \$t7
S0 – S5	Conventional saved registers	<long>	yes	yes	\$s0 – \$s5
S6	Stack frame base register	<long>	yes	yes	\$s6
A0 – A5	Argument registers	<long>	yes	yes	\$a0 – \$a5
T8 – T11	Conventional scratch registers	<long>	yes	yes	\$t8 – \$t11
RA	Return Address register	<long>	yes	yes	\$ra
T12	Procedure value register	<long>	yes	yes	\$t12
AT	Volatile scratch register	<long>	yes	yes	\$at
GP	Global pointer register	<long>	yes	yes	\$gp
SP	Stack pointer	<long>	yes	yes	\$sp
ZERO	ReadAsZero/Sink register	<long>	no	yes	\$zero
PC	Program counter	<code>	no	yes	\$pc
FP	Frame pointer. The Frame Pointer is a software register that TotalView maintains; it is not an actual hardware register. TotalView computes the value of FP as part of the stack backtrace.	<long>	no	yes	\$fp

## Alpha Floating-Point Registers

TotalView displays the Alpha floating-point registers in the Stack Frame Pane of the Process Window. Here is a table that describes how TotalView

treats each floating-point register, and the actions you can take with each register.

**Table 32: Alpha Floating-Point Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F1	Floating-point registers (f registers), used singly	<double>	yes	yes	\$f0 – \$f1
F2 – F9	Conventional saved registers	<double>	yes	yes	\$f2 – \$f9
F10 – F15	Conventional scratch registers	<double>	yes	yes	\$f10 – \$f15
F16 – F21	Argument registers	<double>	yes	yes	\$f16 – \$f21
F22 – F30	Conventional scratch registers	<double>	yes	yes	\$f22 – \$f30
F31	ReadAsZero/Sink register	<double>	yes	yes	\$f31
FPCR	Floating-point control register	<long>	yes	no	\$fpcr

## Alpha FPCR Register

For your convenience, TotalView interprets the bit settings of the Alpha FPCR register. You can edit the value of the FPCR and set it to any of the bit settings outlined in the following table.

**Table 33: Alpha FPCR Register Bit Settings**

Value	Bit Setting	Meaning
SUM	0x8000000000000000	Summary bit
DYN=CHOP	0x0000000000000000	Rounding mode — Chopped rounding mode
DYN=MINF	0x0400000000000000	Rounding mode — Negative infinity
DYN=NORM	0x0800000000000000	Rounding mode — Normal rounding
DYN=PINF	0x0c00000000000000	Rounding mode — Positive infinity

Table 33: Alpha FPCR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
IOV	0x0200000000000000	Integer overflow
INE	0x0100000000000000	Inexact result
UNF	0x0080000000000000	Underflow
OVF	0x0040000000000000	Overflow
DZE	0x0020000000000000	Division by zero
INV	0x0010000000000000	Invalid operation

## HP PA-RISC

This section contains the following information:

- PA-RISC General Registers
- PA-RISC Process Status Word
- PA-RISC Floating-Point Registers
- PA-RISC Floating-Point Format

### PA-RISC General Registers

TotalView displays the PA-RISC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register and the actions you take with them.

Table 34: PA-RISC General Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
r0	Always contains zero	<long>	no	no	\$r0
r1-r31	General registers	<long>	yes	yes	\$r1-\$r31
pc	Current instruction pointer	<long>	yes	yes	\$pc
nxtpc	Next instruction pointer	<long>	yes	yes	\$nxtpc
pcs	Current instruction space	<long>	no	no	\$pcs
nxtpcs	Next instruction space	<long>	no	no	\$nxtpcs
psw	Processor status word	<long>	yes	no	\$psw

Table 34: PA-RISC General Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
sar	Shift amount register	<long>	yes	no	\$sar
sr0-sr7	Space registers	<long>	no	no	\$sr0-\$sr7
recov	Recovery counter	<long>	no	no	\$recov
pid1-pid8	Protection IDs	<long>	no	no	\$pid1-\$pid8
ccr	Coprocessor configuration	<long>	no	no	\$ccr
scr	SFU configuration register	<long>	no	no	\$scr
eiem	External interrupt enable mask	<long>	no	no	\$eiem
iir	Interrupt instruction	<long>	no	no	\$iir
isr	Interrupt space	<long>	no	no	\$isr
ior	Interrupt offset	<long>	no	no	\$ior
cr24-cr26	Temporary registers	<long>	no	no	\$cr24-\$cr26
tp	Thread pointer	<long>	yes	yes	\$tp

## PA-RISC Process Status Word

For your convenience, TotalView interprets the bit settings of the PA-RISC Processor Status Word. You can edit the value of this word and set some of the bits listed in the following table.

Table 35: PA-RISC Processor Status Word

Value	Bit Setting	Meaning
W	0x0000000008000000	64-bit addressing enable
E	0x0000000004000000	Little-endian enable
S	0x0000000002000000	Secure interval timer
T	0x0000000001000000	Taken branch flag
H	0x0000000000800000	Higher-privilege transfer trap enable
L	0x0000000000400000	Lower-privilege transfer trap enable
N	0x0000000000200000	Nullify current instruction
X	0x0000000000100000	Data memory break disable
B	0x0000000000080000	Taken branch flag
C	0x0000000000040000	Code address translation enable
V	0x0000000000020000	Divide step correction

Table 35: PA-RISC Processor Status Word (cont.)

Value	Bit Setting	Meaning
M	0x00000000000010000	High-priority machine check mask
O	0x0000000000000080	Ordered references
F	0x0000000000000020	Performance monitor interrupt unmask
R	0x0000000000000010	Recovery counter enable
Q	0x0000000000000008	Interrupt state collection enable
P	0x0000000000000004	Protection identifier validation enable
D	0x0000000000000002	Data address translation enable
I	0x0000000000000001	External interrupt unmask
C/B	0x000000FF0000FF00	Carry/borrow bits

## PA-RISC Floating-Point Registers

The PA-RISC has 32 floating-point registers. The first four are used for status and exception registers. The rest can be addressed as 64-bit doubles, as two 32-bit floats in the right and left sides of the register, or even-odd pairs of registers as 128-bit extended floats.

Table 36: PA-RISC Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
status	Status register	<int>	no	no	\$status
er1-er7	Exception registers	<int>	no	no	\$er1-\$er7
fr4-fr31	Double floating-point registers	<double>	yes	yes	\$fr4-\$fr31
fr4l-fr31l	Left half floating-point registers	<float>	yes	yes	\$fr4l-\$fr31l
fr4r-fr31r	Right half floating-point registers	<float>	yes	yes	\$fr4r-\$fr31r
fr4/fr5-fr30/fr31	Extended floating-point register pairs	<extended>	yes	yes	\$fr4_fr5-\$fr30_fr31

The floating-point status word controls the arithmetic rounding mode, enables user-level traps, enables floating-point exceptions, and indicates the results of comparisons.

**Table 37: Floating-Point Status Word Use**

Type	Value	Meaning
Rounding Mode	0	Round to nearest
	1	Round toward zero
	2	Round toward +infinity
	3	Round toward -infinity
Exception Enable and Exception Flag Bits	V	Invalid operation
	Z	Division by zero
	O	Overflow
	U	Underflow
	I	Inexact result
Comparison Fields	C	Compare bit; contains the result of the most recent queued compare instruction.
	CQ	Compare queue; contains the result of the second-most recent queued compare through the twelfth-most recent queued compare. Each queued compare instruction shifts the CQ field right one bit and copies the C bit into the left-most position.  This field occupies the same bits as the CA field and is undefined after a targeted compare.
	CA	Compare array; an array of seven compare bits, each of which contains the result of the most recent compare instruction targeting that bit.  This field occupies the same bits as the CQ field and is undefined after a queued compare.
Other Flags:	T	Delayed trap
	D	Denormalized as zero

## PA-RISC Floating-Point Format

The PA-RISC processor supports the IEEE floating-point format.

## IBM Power

This second contains the following information:

- Power General Registers
- Power MSR Register
- Power Floating-Point Registers
- Power FPSCR Register
- Using the Power FPSCR Register

**NOTE** The Power architecture supports the IEEE floating-point format.

### Power General Registers

TotalView displays Power general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

**Table 38: Power General Purpose Integer Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
R0	General register 0	<int>	yes	yes	\$r0
SP	Stack pointer	<int>	yes	yes	\$sp
RTOC	TOC pointer	<int>	yes	yes	\$rtoc
R3 – R31	General registers 3 – 31	<int>	yes	yes	\$r3 – \$r31
INUM		<int>	yes	no	\$inum
PC	Program counter	<code>	no	yes	\$pc
SRR1	Machine status save/restore register	<int>	yes	no	\$srr1
LR	Link register	<int>	yes	no	\$lr
CTR	Counter register	<int>	yes	no	\$ctr
CR	Condition register	<int>	yes	no	\$cr
XER	Integer exception register	<int>	yes	no	\$xer
DAR	Data address register	<int>	yes	no	\$dar
MQ	MQ register	<int>	yes	no	\$mq
MSR	Machine state register	<int>	yes	no	\$msr
SEG0 – SEG9	Segment registers 0 – 9	<int>	yes	no	\$seg0 – \$seg9



Table 38: Power General Purpose Integer Registers (cont.)

Register	Description	Data Type	Edit	Dive	Specify in Expression
SG10 – SG15	Segment registers 10 –15	<int>	yes	no	\$sg10 – \$sg15
SCNT	SS_COUNT	<int>	yes	no	\$scnt
SAD1	SS_ADDR 1	<int>	yes	no	\$sad1
SAD2	SS_ADDR 2	<int>	yes	no	\$sad2
SCD1	SS_CODE 1	<int>	yes	no	\$scd1
SCD2	SS_CODE 2	<int>	yes	no	\$scd2
TID		<int>	yes	no	

## Power MSR Register

For your convenience, TotalView interprets the bit settings of the Power MSR register. You can edit the value of the MSR and set it to any of the bit settings outlined in the following table.

Table 39: Power MSR Register Bit Settings

Value	Bit Setting	Meaning
0x00040000	POW	Power management enable
0x00020000	TGPR	Temporary GPR mapping
0x00010000	ILE	Exception little-endian mode
0x00008000	EE	External interrupt enable
0x00004000	PR	Privilege level
0x00002000	FP	Floating-point available
0x00001000	ME	Machine check enable
0x00000800	FE0	Floating-point exception mode 0
0x00000400	SE	Single-step trace enable
0x00000200	BE	Branch trace enable
0x00000100	FE1	Floating-point exception mode 1
0x00000040	IP	Exception prefix
0x00000020	IR	Instruction address translation
0x00000010	DR	Data address translation
0x00000002	RI	Recoverable exception
0x00000001	LE	Little-endian mode enable

## Power Floating-Point Registers

TotalView displays the Power floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

**Table 40: Power Floating-Point Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0 – F31	Floating-point registers 0 – 31	<double>	yes	yes	\$f0 – \$f31
FPSCR	Floating-point status register	<int>	yes	no	\$fpscr
FPSCR2	Floating-point status register 2	<int>	yes	no	\$fpscr2

## Power FPSCR Register

For your convenience, TotalView interprets the bit settings of the Power FPSCR register. You can edit the value of the FPSCR and set it to any of the bit settings outlined in the following table.

**Table 41: Power PFSCR Register Bit Settings**

Value	Bit Setting	Meaning
0x80000000	FX	Floating-point exception summary
0x40000000	FEX	Floating-point enabled exception summary
0x20000000	VX	Floating-point invalid operation exception summary
0x10000000	OX	Floating-point overflow exception
0x08000000	UX	Floating-point underflow exception
0x04000000	ZX	Floating-point zero divide exception
0x02000000	XX	Floating-point inexact exception
0x01000000	VXSNAN	Floating-point invalid operation exception for SNaN
0x00800000	VXISI	Floating-point invalid operation exception: $\infty - \infty$

Table 41: Power PFSCR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
0x00400000	VXIDI	Floating-point invalid operation exception: $\infty / \infty$
0x00200000	VXZDZ	Floating-point invalid operation exception: 0 / 0
0x00100000	VXIMZ	Floating-point invalid operation exception: $\infty * \infty$
0x00080000	VXVC	Floating-point invalid operation exception: invalid compare
0x00040000	FR	Floating-point fraction rounded
0x00020000	FI	Floating-point fraction inexact
0x00010000	FPRF=(C)	Floating-point result class descriptor
0x00008000	FPRF=(L)	Floating-point less than or negative
0x00004000	FPRF=(G)	Floating-point greater than or positive
0x00002000	FPRF=(E)	Floating-point equal or zero
0x00001000	FPRF=(U)	Floating-point unordered or NaN
0x00011000	FPRF=(QNAN)	Quiet NaN; alias for FPRF=(C+U)
0x00009000	FPRF=(-INF)	-Infinity; alias for FPRF=(L+U)
0x00008000	FPRF=(-NORM)	-Normalized number; alias for FPRF=(L)
0x00018000	FPRF=(-DENORM)	-Denormalized number; alias for FPRF=(C+L)
0x00012000	FPRF=(-ZERO)	-Zero; alias for FPRF=(C+E)
0x00002000	FPRF=(+ZERO)	+Zero; alias for FPRF=(E)
0x00014000	FPRF=(+DENORM)	+Denormalized number; alias for FPRF=(C+G)
0x00004000	FPRF=(+NORM)	+Normalized number; alias for FPRF=(G)
0x00005000	FPRF=(+INF)	+Infinity; alias for FPRF=(G+U)
0x00000400	VXSOF	Floating-point invalid operation exception: software request
0x00000200	VXSQRT	Floating-point invalid operation exception: square root
0x00000100	VXCVI	Floating-point invalid operation exception: invalid integer convert
0x00000080	VE	Floating-point invalid operation exception enable
0x00000040	OE	Floating-point overflow exception enable
0x00000020	UE	Floating-point underflow exception enable

Table 41: Power FPSCR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
0x00000010	ZE	Floating-point zero divide exception enable
0x00000008	XE	Floating-point inexact exception enable
0x00000004	NI	Floating-point non-IEEE mode enable
0x00000000	RN=NEAR	Round to nearest
0x00000001	RN=ZERO	Round toward zero
0x00000002	RN=PINF	Round toward +infinity
0x00000003	RN=NINF	Round toward -infinity

## Using the Power FPSCR Register

On AIX, if you compile your program to catch floating-point exceptions (IBM compiler **-qflttrap** option), you can change the value of the FPSCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSCR to include **0x10** (as shown in Table 41) so that TotalView traps the "divide by zero" exception. The string displayed next to the FPSR register should now include **ZE**. Now, when your program divides by zero, it receives a **SIGTRAP** signal, which will be caught by TotalView. See Chapter 3 "Setting Up a Debugging Session" on page 33 and "Handling Signals" on page 45 for more information. If you did not set the bit for trapping divide by zero or you did not compile to catch floating-point exceptions, your program would not stop and the processor would set the **ZX** bit.

## Intel-x86

This section contains the following information:

- Intel-x86 General Registers
- Intel-x86 Floating-Point Registers
- Intel-x86 FPCR Register
- Using the Intel-x86 FPCR Register

## ■ Intel-x86 FPSR Register

**NOTE** The Intel-x86 processor supports the IEEE floating-point format.

## Intel-x86 General Registers

TotalView displays the Intel-x86 general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

**Table 42: Intel-x86 General Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
EAX	General registers	<void>	yes	yes	\$eax
ECX		<void>	yes	yes	\$ecx
EDX		<void>	yes	yes	\$edx
EBX		<void>	yes	yes	\$ebx
EBP		<void>	yes	yes	\$ebp
ESP		<void>	yes	yes	\$esp
ESI		<void>	yes	yes	\$esi
EDI		<void>	yes	yes	\$edi
CS	Selector registers	<void>	no	no	\$cs
SS		<void>	no	no	\$ss
DS		<void>	no	no	\$ds
ES		<void>	no	no	\$es
FS		<void>	no	no	\$fs
GS		<void>	no	no	\$gs
EFLAGS	Instruction pointer	<void>	no	no	\$eflags
EIP		<code>	no	yes	\$eip
FAULT		<void>	no	no	\$fault
TEMP		<void>	no	no	\$temp
INUM		<void>	no	no	\$inum
ECODE		<void>	no	no	\$ecode

## Intel-x86 Floating-Point Registers

TotalView displays the x86 floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

**Table 43: Intel-x86 Floating-Point Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
ST0	ST(0)	<extended>	yes	yes	\$st0
ST1	ST(1)	<extended>	yes	yes	\$st1
ST2	ST(2)	<extended>	yes	yes	\$st2
ST3	ST(3)	<extended>	yes	yes	\$st3
ST4	ST(4)	<extended>	yes	yes	\$st4
ST5	ST(5)	<extended>	yes	yes	\$st5
ST6	ST(6)	<extended>	yes	yes	\$st6
ST7	ST(7)	<extended>	yes	yes	\$st7
FPCR	Floating-point control register	<void>	yes	no	\$fpcr
FPSR	Floating-point status register	<void>	no	no	\$fpsr
FPTAG	Tag word	<void>	no	no	\$fptag
FPIOFF	Instruction offset	<void>	no	no	\$fpioff
FPISEL	Instruction selector	<void>	no	no	\$fpisel
FPDOFF	Data offset	<void>	no	no	\$fpdoff
FPDSEL	Data selector	<void>	no	no	\$fpdsel

## Intel-x86 FPCR Register

For your convenience, TotalView interprets the bit settings of the FPCR and FPSR registers.

You can edit the value of the FPCR and set it to any of the bit settings outlined in the next table.

**Table 44: Intel-x86 FPCR Register Bit Settings**

Value	Bit Setting	Meaning
RC=NEAR	0x0000	To nearest rounding mode
RC=NINF	0x0400	Toward negative infinity rounding mode
RC=PINF	0x0800	Toward positive infinity rounding mode
RC=ZERO	0x0c00	Toward zero rounding mode
PC=SGL	0x0000	Single-precision rounding
PC=DBL	0x0080	Double-precision rounding
PC=EXT	0x00c0	Extended-precision rounding
EM=PM	0x0020	Precision exception enable
EM=UM	0x0010	Underflow exception enable
EM=OM	0x0008	Overflow exception enable
EM=ZM	0x0004	Zero-divide exception enable
EM=DM	0x0002	Denormalized operand exception enable
EM=IM	0x0001	Invalid operation exception enable

## Using the Intel-x86 FPCR Register

You can change the value of the FPCR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPCR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPCR to include **0x0004** (as shown in Table 44) so that TotalView traps the “divide-by-zero” bit. The string displayed next to the FPCR register should now include **EM=(ZM)**. Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See “*Handling Signals*” on page 45 for information on handling signals. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the **EF=(ZE)** bit in the FPSR.

## Intel-x86 FPSR Register

The bit settings of the Intel-x86 FPSR register are outlined in the following table.

**Table 45: Intel-x86 FPSR Register Bit Settings**

Value	Bit Setting	Meaning
TOP=< <i>i</i> >	0x3800	Register < <i>i</i> > is top of FPU stack
B	0x8000	FPU busy
C0	0x0100	Condition bit 0
C1	0x0200	Condition bit 1
C2	0x0400	Condition bit 2
C3	0x4000	Condition bit 3
ES	0x0080	Exception summary status
SF	0x0040	Stack fault
EF=PE	0x0020	Precision exception
EF=UE	0x0010	Underflow exception
EF=OE	0x0008	Overflow exception
EF=ZE	0x0004	Zero divide exception
EF=DE	0x0002	Denormalized operand exception
EF=IE	0x0001	Invalid operation exception

## SGI MIPS

This section contains the following information:

- MIPS General Registers
- MIPS SR Register
- MIPS Floating-Point Registers
- MIPS FCSR Register
- Using the MIPS FCSR Register
- MIPS Delay Slot Instructions

**NOTE** The MIPS processor supports the IEEE floating-point format.



## MIPS General Registers

TotalView displays the MIPS general purpose registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Programs compiled with either **-64** or **-n32** have 64-bit registers. TotalView uses **<long>** for **-64** compiled programs and **<long long>** for **-n32** compiled programs.

**Table 46: MIPS General (Integer) Registers**

Register	Description	Data Type	Edit	Dive	Specify in Expression
ZERO	Always has the value 0	<long>	no	no	\$zero
AT	Reserved for the assembler	<long>	yes	yes	\$at
V0 – V1	Function value registers	<long>	yes	yes	\$v0 – \$v1
A0 – A7	Argument registers	<long>	yes	yes	\$a0 – \$a7
T0 – T3	Temporary registers	<long>	yes	yes	\$t0 – \$t3
S0 – S7	Saved registers	<long>	yes	yes	\$s0 – \$s7
T8 – T9	Temporary registers	<long>	yes	yes	\$t8 – \$t9
K0 – K1	Reserved for the operating system	<long>	yes	yes	\$k1 – \$k2
GP	Global pointer	<long>	yes	yes	\$gp
SP	Stack pointer	<long>	yes	yes	\$sp
S8	Hardware frame pointer	<long>	yes	yes	\$s8
RA	Return address register	<code>	no	yes	\$ra
MDLO	Multiply/Divide special register, holds least-significant bits of multiply, quotient of divide	<long>	yes	yes	\$mdlo
MDHI	Multiply/Divide special register, holds most-significant bits of multiply, remainder of divide	<long>	yes	yes	\$mdhi
CAUSE	Cause register	<long>	yes	yes	\$cause
EPC	Program counter	<code>	no	yes	\$epc

Table 46: MIPS General (Integer) Registers (cont.)

Register	Description	Data Type	Edit	Dive	Specify in Expression
SR	Status register	<long>	no	no	\$sr
VFP	Virtual frame pointer	<long>	no	no	\$vfp
	The virtual frame pointer is a software register that TotalView maintains. It is not an actual hardware register. TotalView computes the VFP as part of stack backtrace.				

## MIPS SR Register

For your convenience, TotalView interprets the bit settings of the SR register as outlined in the next table.

Table 47: MIPS SR Register Bit Settings

Value	Bit Setting	Meaning
0x00000001	IE	Interrupt enable
0x00000002	EXL	Exception level
0x00000004	ERL	Error level
0x00000008	S	Supervisor mode
0x00000010	U	User mode
0x00000018	U	Undefined (implemented as User mode)
0x00000000	K	Kernel mode
0x00000020	UX	User mode 64-bit addressing
0x00000040	SX	Supervisor mode 64-bit addressing
0x00000080	KX	Kernel mode 64-bit addressing
0x0000FF00	IM= <i>i</i>	Interrupt Mask value is <i>i</i>
0x00010000	DE	Disable cache parity/ECC
0x00020000	CE	Reserved
0x00040000	CH	Cache hit
0x00080000	NMI	Non-maskable interrupt has occurred
0x00100000	SR	Soft reset or NMI exception
0x00200000	TS	TLB shutdown has occurred
0x00400000	BEV	Bootstrap vectors

Table 47: MIPS SR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
0x02000000	RE	Reverse-Endian bit
0x04000000	FR	Additional floating-point registers enabled
0x08000000	RP	Reduced power mode
0x10000000	CU0	Coprocessor 0 usable
0x20000000	CU1	Coprocessor 1 usable
0x40000000	CU2	Coprocessor 2 usable
0x80000000	XX	MIPS IV instructions usable

## MIPS Floating-Point Registers

TotalView displays the MIPS floating-point registers in the Stack Frame Pane of the Process Window. Here is a table that describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 48: MIPS Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0, F2	Hold results of floating-point type function; \$f0 has the real part, \$f2 has the imaginary part	<double>	yes	yes	\$f0, \$f2
F1 – F3, F4 – F11	Temporary registers	<double>	yes	yes	\$f1 – \$f3, \$f4 – \$f11
F12 – F19	Pass single- or double-precision actual arguments	<double>	yes	yes	\$f12 – \$f19
F20 – F23	Temporary registers	<double>	yes	yes	\$f20 – \$f23
F24 – F31	Saved registers	<double>	yes	yes	\$f24 – \$f31
FCSR	FPU control and status register	<int>	yes	no	\$fcsr

## MIPS FCSR Register

For your convenience, TotalView interprets the bit settings of the MIPS FCSR register. You can edit the value of the FCSR and set it to any of the bit settings outlined in the following table.

**Table 49: MIPS FCSR Register Bit Settings**

Value	Bit Setting	Meaning
RM=RN	0x00000000	Round to nearest
RM=RZ	0x00000001	Round toward zero
RM=RP	0x00000002	Round toward positive infinity
RM=RM	0x00000003	Round toward negative infinity
flags=(I)	0x00000004	Flag=inexact result
flags=(U)	0x00000008	Flag=underflow
flags=(O)	0x00000010	Flag=overflow
flags=(Z)	0x00000020	Flag=divide by zero
flags=(V)	0x00000040	Flag=invalid operation
enables=(I)	0x00000080	Enables=inexact result
enables=(U)	0x00000100	Enables=underflow
enables=(O)	0x00000200	Enables=overflow
enables=(Z)	0x00000400	Enables=divide by zero
enables=(V)	0x00000800	Enables=invalid operation
cause=(I)	0x00001000	Cause=inexact result
cause=(U)	0x00002000	Cause=underflow
cause=(O)	0x00004000	Cause=overflow
cause=(Z)	0x00008000	Cause=divide by zero
cause=(V)	0x00010000	Cause=invalid operation
cause=(E)	0x00020000	Cause=unimplemented
FCC=(0/c)	0x00800000	FCC=Floating-Point Condition Code 0; c=Condition bit
FS	0x01000000	Flush to zero
FCC=(1)	0x02000000	FCC=Floating-Point Condition Code 1
FCC=(2)	0x04000000	FCC=Floating-Point Condition Code 2
FCC=(3)	0x08000000	FCC=Floating-Point Condition Code 3
FCC=(4)	0x10000000	FCC=Floating-Point Condition Code 4
FCC=(5)	0x20000000	FCC=Floating-Point Condition Code 5

Table 49: MIPS FCSR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
FCC=(6)	0x40000000	FCC=Floating-Point Condition Code 6
FCC=(7)	0x80000000	FCC=Floating-Point Condition Code 7

## Using the MIPS FCSR Register

You can change the value of the MIPS FCSR register within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FCSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FCSR to include **0x400** (as shown in Table 49). The string displayed next to the FCSR register should now include **enables=(Z)**. Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See Chapter 3 “*Setting Up a Debugging Session*” on page 33 and “*Handling Signals*” on page 45 for more information.

## MIPS Delay Slot Instructions

On the MIPS architecture, jump and branch instructions have a “delay slot”. This means that the instruction after the jump or branch instruction is executed before the jump or branch is executed.

In addition, there is a group of “branch likely” conditional branch instructions in which the instruction in the delay slot is executed only if the branch is taken.

The MIPS processors execute the jump or branch instruction and the delay slot instruction as an indivisible unit. If an exception occurs as a result of executing the delay slot instruction, the branch or jump instruction is not executed, and the exception appears to have been caused by the jump or branch instruction.

This behavior of the MIPS processors affects both the TotalView instruction step command and TotalView breakpoints.

The TotalView instruction step command will step both the jump or branch instruction and the delay slot instruction as if they were a single instruction.

If a breakpoint is placed on a delay slot instruction, execution will stop at the jump or branch preceding the delay slot instruction, and TotalView will not know that it is at a breakpoint. At this point, attempting to continue the thread that hit the breakpoint without first removing the breakpoint will cause the thread to hit the breakpoint again without executing any instructions. Before continuing the thread, you must remove the breakpoint. If you need to reestablish the breakpoint, you might then use the instruction step command to execute just the delay slot instruction and the branch.

A breakpoint placed on a delay slot instruction of a *branch likely* instruction will be hit only if the branch is going to be taken.

## Sun SPARC

This section has the following information:

- SPARC General Registers
- SPARC PSR Register
- SPARC Floating-Point Registers
- SPARC FPSR Register
- Using the SPARC FPSR Register

**NOTE** The SPARC processor supports the IEEE floating-point format.

## SPARC General Registers

TotalView displays the SPARC general registers in the Stack Frame Pane of the Process Window. The following table describes how TotalView treats each general register, and the actions you can take with each register.

Table 50: SPARC General Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
G0	Global zero register	<int>	no	no	\$g0
G1 – G7	Global registers	<int>	yes	yes	\$g1 – \$g7
O0 – O5	Outgoing parameter registers	<int>	yes	yes	\$o0 – \$o5
SP	Stack pointer	<int>	yes	yes	\$sp
O7	Temporary register	<int>	yes	yes	\$o7
L0 – L7	Local registers	<int>	yes	yes	\$l0 – \$l7
I0 – I5	Incoming parameter registers	<int>	yes	yes	\$i0 – \$i5
FP	Frame pointer	<int>	yes	yes	\$fp
I7	Return address	<int>	yes	yes	\$i7
PSR	Processor status register	<int>	yes	no	\$psr
Y	Y register	<int>	yes	yes	\$y
WIM	WIM register	<int>	no	no	
TBR	TBR register	<int>	no	no	
PC	Program counter	<code>	no	yes	\$pc
nPC	Next program counter	<code>	no	yes	\$npc

## SPARC PSR Register

For your convenience, TotalView interprets the bit settings of the SPARC PSR register. You can edit the value of the PSR and set some of the bits outlined in the following table.

Table 51: SPARC PSR Register Bit Settings

Value	Bit Setting	Meaning
ET	0x00000020	Traps enabled
PS	0x00000040	Previous supervisor

Table 51: SPARC PSR Register Bit Settings

Value	Bit Setting	Meaning
S	0x00000080	Supervisor mode
EF	0x00001000	Floating-point unit enabled
EC	0x00002000	Coprocessor enabled
C	0x00100000	Carry condition code
V	0x00200000	Overflow condition code
Z	0x00400000	Zero condition code
N	0x00800000	Negative condition code

## SPARC Floating-Point Registers

TotalView displays the SPARC floating-point registers in the Stack Frame Pane of the Process Window. The next table describes how TotalView treats each floating-point register, and the actions you can take with each register.

Table 52: SPARC Floating-Point Registers

Register	Description	Data Type	Edit	Dive	Specify in Expression
F0, F1, F0_F1	Floating-point registers (f registers), used singly	<float>	no	yes	\$f0, \$f1, \$f0_f1
F2 – F31	Floating-point registers (f registers), used singly	<float>	yes	yes	\$f2– \$f31
F0, F1, F0_F1	Floating-point registers (f registers), used as pairs	<double>	no	yes	\$f0, \$f1, \$f0_f1
F0/F1 – F30/F31	Floating-point registers (f registers), used as pairs	<double>	yes	yes	\$2 – \$f30_f31
FPCR	Floating-point control register	<int>	no	no	\$fpcr
FPSR	Floating-point status register	<int>	yes	no	\$fpsr

TotalView allows you to use these registers singly or in pairs, depending on how they are used by your program. For example, if you use F1 by itself, its type is **<float>**, but if you use the F0/F1 pair, its type is **<double>**.



## SPARC FPSR Register

For your convenience, TotalView interprets the bit settings of the SPARC FPSR register. You can edit the value of the FPSR and set it to any of the bit settings outlined in the following table.

**Table 53: SPARC FPSR Register Bit Settings**

Value	Bit Setting	Meaning
CEXC=NX	0x00000001	Current inexact exception
CEXC=DZ	0x00000002	Current divide by zero exception
CEXC=UF	0x00000004	Current underflow exception
CEXC=OF	0x00000008	Current overflow exception
CEXC=NV	0x00000010	Current invalid exception
AEXC=NX	0x00000020	Accrued inexact exception
AEXC=DZ	0x00000040	Accrued divide by zero exception
AEXC=UF	0x00000080	Accrued underflow exception
AEXC=OF	0x00000100	Accrued overflow exception
AEXC=NV	0x00000200	Accrued invalid exception
EQ	0x00000000	Floating-point condition =
LT	0x00000400	Floating-point condition <
GT	0x00000800	Floating-point condition >
UN	0x00000c00	Floating-point condition unordered
QNE	0x00002000	Queue not empty
NONE	0x00000000	Floating-point trap type None
IEEE	0x00004000	Floating-point trap type IEEE Exception
UFIN	0x00008000	Floating-point trap type Unfinished FPop
UIMP	0x0000c000	Floating-point trap type Unimplemented FPop
SEQE	0x00010000	Floating-point trap type Sequence Error
NS	0x00400000	Nonstandard floating-point FAST mode
TEM=NX	0x00800000	Trap enable mask – Inexact Trap Mask
TEM=DZ	0x01000000	Trap enable mask – Divide by Zero Trap Mask
TEM=UF	0x02000000	Trap enable mask – Underflow Trap Mask
TEM=OF	0x04000000	Trap enable mask – Overflow Trap Mask
TEM=NV	0x08000000	Trap enable mask – Invalid Operation Trap Mask
EXT	0x00000000	Extended rounding precision – Extended precision

Table 53: SPARC FPSR Register Bit Settings (cont.)

Value	Bit Setting	Meaning
SGL	0x10000000	Extended rounding precision – Single precision
DBL	0x20000000	Extended rounding precision – Double precision
NEAR	0x00000000	Rounding direction – Round to nearest (tie-even)
ZERO	0x40000000	Rounding direction – Round to 0
PINF	0x80000000	Rounding direction – Round to +Infinity
NINF	0xc0000000	Rounding direction – Round to -Infinity

## Using the SPARC FPSR Register

The SPARC processor does not catch floating-point errors by default. You can change the value of the FPSR within TotalView to customize the exception handling for your program.

For example, if your program inadvertently divides by zero, you can edit the bit setting of the FPSR register in the Stack Frame Pane. In this case, you would change the bit setting for the FPSR to include **0x01000000** (as shown in Table 53) so that TotalView traps the "divide by zero" bit. The string displayed next to the FPSR register should now include **TEM=(DZ)**. Now, when your program divides by zero, it receives a **SIGFPE** signal, which you can catch with TotalView. See Chapter 3 "Setting Up a Debugging Session" on page 33 and "Handling Signals" on page 45 for more information. If you did not set the bit for trapping divide by zero, the processor would ignore the error and set the **AEXC=(DZ)** bit.



# Glossary

**ACTION POINT:** A debugger feature that allows a user to request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, evaluation points, and barriers.

**ACTION POINT IDENTIFIER:** A unique integer ID associated with an action point.

**ADDRESS SPACE:** A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

**AFFECTED P/T SET:** The set of process and threads that will be affected by the command. For most commands, this is identical to the target P/T set, but in some cases it may include additional threads. (See "P/T (process/thread) set" on page 371 for more information.)

**AGGREGATED OUTPUT:** The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

**ARENA:** A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

**ASYNCHRONOUS:** When processes communicate with one another, they send messages. If a process decides that it does not want to wait for an answer, it is said to run "asynchronously". For example, in most client/server programs, one program sends an RPC request to a second program and then waits to receive a response from the second program. This is the normal *synchronous* mode of operation. If, however, the first program

sends a message and then continues executing, not waiting for a reply, the first mode of operations is said to be *asynchronous*.

**AUTOMATIC PROCESS ACQUISITION:** TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you do not have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition automatically starts the TotalView Debugger Server (the **tvdsvr**).

**BARRIER:** An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

**BASE WINDOW:** The original Process Window or Variable Window before you dive into routines or variables. After diving, you can use a **Reset** or **Undive** command to restore this original window.

**BLOCKED:** A thread state where the thread is no longer executing because it is waiting for an event to occur. In most cases, the thread is blocked because it is waiting for a mutex or condition state.

**BREAKPOINT:** A point in a program where execution can be suspended to permit examination and manipulation of data.

**CALL STACK:** A higher-level view of stack memory, interpreted in terms of source program variables and locations.

**CHILD PROCESS:** A process created by another process (see "parent process" on page 370) when that other process calls **fork()**.

**CLOSED LOOP:** see **closed loop**.

**CLUSTER DEBUGGING:** The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

**COMMAND HISTORY LIST:** A debugger-maintained list storing copies of the most recent commands issued by the user.

**CONDITION SYNCHRONIZATON:** A process that delays thread execution until a condition is satisfied.

**CONTEXTUALLY QUALIFIED (SYMBOL):** A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

**CORE FILE:** A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

**CORE-FILE DEBUGGING:** A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

**CROSS-DEBUGGING:** A special case of remote debugging where the host platform and the target platform are different types of machines.

**CURRENT FRAME:** The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

**CURRENT LANGUAGE:** The source code language used by the file containing the current source location.

**CURRENT LIST LOCATION:** The location governing what source code will be displayed in response to a list command.

**DATASET:** A set of array elements generated by TotalView and sent to the Visualizer. (See "**visualizer process**" on page 376.)

**DBELOG LIBRARY:** A library of routines for creating event points and generating event logs from within TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

**DBFORK LIBRARY:** A library of special versions of the **fork()** and **execve()** calls used by the TotalView debugger to debug multiprocess programs. If you link your program with TotalView's **dbfork** library, TotalView will be able to automatically attach to newly spawned processes.

**DEBUGGING INFORMATION:** Information relating an executable to the source code from which it was generated.

**DEBUGGER INITIALIZATION FILE:** An optional file establishing initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called `.tvdrc`.

**DEBUGGER PROMPT:** A string printed by the CLI that indicates that it is ready to receive another user command.

**DEBUGGER SERVER:** See **tvdsvr process**.

**DEBUGGER STATE:** Information that TotalView or the CLI maintains in order to interpret and respond to user commands. Includes debugger modes, user-defined commands, and debugger variables.

**DISTRIBUTED DEBUGGING:** The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PAR-MACS) run on more than one host.

**DIVE STACK:** A series of nested dives that were performed in the same variable window. The number of greater-than symbols (>) in the upper left-hand corner of a Variable Window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of greater-than symbols shown in the Variable Window.

**DIVING:** The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

**DOPE VECTOR:** This is a runtime descriptor that contains all information about an object that requires more information than is available as a single pointer or value. For example, you might declare a Fortran 90 pointer variable that is a pointer to some other object but which has its own upper bound as follows:

`integer, pointer, dimension (:) :: iptr`

Assume that you initialize it as follows:

`iptr => iarray(20:1:-2)`

`iptr` is now a synonym for every other element in the first twenty elements of `iarray` and this pointer array is in reverse order. For example, `iptr(1)` maps to `iarray(20)`, `iptr(2)` maps to `iarray(18)`, and so on.

A compiler represents an `iptr` object using a run time descriptor) that contains (at least) elements such as a pointer to the first element of the actual data, a stride value, and a count of the number of elements (or equivalently an upper bound).

**DPID:** Debugger ID. This is the ID TotalView uses for processes.

**EDITING CURSOR:** A black rectangle that appears when a TotalView GUI field is selected for editing. You use field editor commands to move the editing cursor.

**EVALUATION POINT:** A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

**EVENT LOG:** A file containing a record of events for each process in a program.

**EVENT POINT:** A point in the program where TotalView writes an event to the event log for later analysis with TimeScan.

**EXECUTABLE:** A compiled and linked version of source files, containing a "main" entry point.

**EXPRESSION:** An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current source language. Not all Fortran 90, C, and C++ operators are supported.

**EXTENT:** The number of elements in the dimension of an array. For example, a Fortran array of `integer(7,8)` has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

**FIELD EDITOR:** A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.

**FOCUS:** The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you are using the default prompt).

**FRAME:** An area in stack memory containing the information corresponding to a single invocation of a subprocedure.

**FULLY QUALIFIED (SYMBOL):** A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

**GID:** The TotalView group ID.

**GOI:** The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and the like.

**GRIDGET:** A dotted grid in the tag field that indicates you can set an action point on the instruction.

**GROUP:** When TotalView starts processes, it places related processes in families. These families are called "groups."

**GROUP OF INTEREST:** The primary group that is affected by a command.

**HEAP:** An area of memory that your program uses when it dynamically allocates blocks of memory. It is also how people describe my car.

**HOST MACHINE:** The machine on which the TotalView debugger is running.

**INITIAL PROCESS:** The process created as part of a load operation, or that already existed in the run-time environment and was attached by TotalView or the CLI.

**INFINITE LOOP:** See **loop**, **infinite**.

**LVALUE:** A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

**LHS EXPRESSION:** This is a synonym for **lvalue**.

**LOOP, INFINITE:** see **infinite loop**.



**LOWER BOUND:** The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

**MACHINE STATE:** Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

**MESSAGE QUEUE:** A list of messages sent and received by message-passing programs.

**MPICH:** MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see [www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi).

**MPMD (MULTIPLE PROGRAM MULTIPLE DATA) PROGRAMS:** A program involving multiple executables, executed by multiple threads and processes.

**MUTEX (MUTUAL EXCLUSION):** Techniques for sharing resources so that different users do not conflict and cause unwanted interactions.

**NATIVE DEBUGGING:** The action of debugging a program that is running on the same machine as TotalView.

**NESTED DIVE:** TotalView lets you dive into pointers, structures, or arrays within a variable. When you dive into one of these elements, TotalView updates the display so that the new element is displayed. So, a nested dive is a *dive* within a dive. You can return to the previous display by selecting the left-facing arrow in the top right corner of the window.

**NODE:** A machine on a network. Each machine has a unique network name and address.

**OUT OF SCOPE:** When symbol lookup is performed for a particular symbol name and it is not found in the current scope or any containing scopes, the symbol is said to be out of scope.

**PARALLEL PROGRAM:** A program whose execution involves multiple threads and processes.

**PARALLEL TASKS:** Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results. (llnl)

**PARALLELIZABLE PROBLEM:** A problem that can be divided into parallel tasks. This may require changes in the code and/or the underlying algorithm. (llnl)

**PARCEL:** The number of bytes required to hold the shortest instruction for the target architecture.

**PARENT PROCESS:** A process that calls `fork()` to spawn other processes (usually called "child processes").

**PARMACS LIBRARY:** A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

**PARTIALLY QUALIFIED (SYMBOL):** A symbol name that includes only some of the levels of source code organization (for example, filename and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

**PC:** This is an abbreviation for *Program Counter*.

**PID:** Depending on context, this is either the "process ID" or the "program ID". In most cases, this will be a process ID.

**POI:** The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and the like.

**PROCESS:** An executable that is loaded into memory and is running (or capable of running).

**PROCESS GROUP:** A group of processes associated with a multiprocess program. A process group includes program control groups and share groups.

**PROCESS/THREAD IDENTIFIER:** A unique integer ID associated with a particular process and thread.

**PROCESS OF INTEREST:** The primary process that is affected by a command.

**PROGRAM EVENT:** A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

**PROGRAM CONTROL GROUP:** A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to `execve()` (processes that do *not* share the same source code as the parent). Contrast with **share group**.

**PROGRAM STATE:** A higher-level view of the machine state, where addresses, instructions, registers, and such, are interpreted in terms of source program variables and statements.

**P/T (PROCESS/THREAD) SET:** The set of threads drawn from all threads in all processes of the target program.

**PVM LIBRARY:** Parallel Virtual Machine library. A message-passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

**RACE CONDITION:** A problem that occurs when threads try to simultaneously access a resource. The result can be a deadlock, data corruption, or a program fault.

**REMOTE DEBUGGING:** The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

**RESUME COMMANDS:** Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

**RHS EXPRESSION:** This is a synonym for **rvalue**.

**RVALUE:** An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

**SATISFACTION SET:** The set of processes and threads that must be held before a barrier can be satisfied.

**SATISFIED:** A condition indicating that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is *satisfied*, the held processes or threads are released, which means they can now be run. Prior to this event, they could not be run.

**SERIAL EXECUTION:** Execution of a program sequentially, one statement at a time. (*llnl*)

**SERIAL LINE DEBUGGING:** A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

**SHARE GROUP:** A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with **program control group**.

**SHARED LIBRARY:** A compiled and linked set of source files that are dynamically loaded by other executables—and have no “main” entry point.

**SIGNALS:** Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

**SINGLE STEP:** The action of executing a single statement and stopping (as if at a breakpoint).

**SLICE:** A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.

**SOURCE FILE:** Program file containing source language statements. TotalView allows you to debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler.

**SOURCE LOCATION:** For each thread, the source code line it will execute next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

**SPAWNED PROCESS:** The process created by a user process executing under debugger control.

**SPMD (SINGLE PROGRAM MULTIPLE DATA) PROGRAMS:** A program involving just one executable, executed by multiple threads and processes.

**STACK:** A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

**STACK FRAME:** A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the program counter (PC) at the time the routine was called.

**STACK POINTER:** A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored.

**STACK TRACE:** A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.

**STATIC (SYMBOL) SCOPE:** A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

**STEPPING:** Advancing program execution by fixed increments, such as by source code statements.

**STOP SET:** A set of threads that should be stopped once an action point has been triggered.

**STOPPED/HELD STATE:** The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) will be required before it is capable of continuing execution.

**STOPPED/RUNNABLE STATE:** The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

**STOPPED STATE:** The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

**STRIDE:** The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is -1, every element between the upper bound and lower bound (reverse order) is displayed.

**SYMBOL:** Entities within program state, machine state, or debugger state.

**SYMBOL LOOKUP:** Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively so that containing scopes are searched in an outward progression.

**SYMBOL NAME:** The name associated with a symbol known to TotalView (for example, function, variable, data type, and such).

**SYMBOL TABLE:** A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **-g** option) and is used by debuggers to analyze the program.

**SYNCHRONIZATION:** A mechanism that prevents problems caused by concurrent threads manipulating shared resources. The two most common mechanisms for synchronizing threads are mutual exclusion and condition synchronization.

**TAG FIELD:** The left margin in the Source Pane of the TotalView Process Window containing boxed line numbers marking the lines of source code that actually generate executable code.

**TARGET MACHINE:** The machine on which the process to be debugged is running.

**TARGET PROCESS SET:** The target set for those occasions when operations can only be applied to entire processes, not to individual threads within a process.

**TARGET PROGRAM:** The executing program that is the target of debugger operations.

**TARGET P/T SET:** The set of processes and threads upon which a CLI command will act.

**TASK:** A logically discrete section of computational work. (This is an informal definition.) (*llnl*)

**THREAD:** An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

**THREAD EXECUTION STATE:** The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of predefined states.

**THREAD OF INTEREST:** The primary thread that will be affected by a command.

**TID:** The thread ID.

**TOI:** The thread of interest. This is the primary thread that will be affected by a command.

**TRIGGER SET:** The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

**TRIGGERS:** The effect during execution when program operations cause an event to occur (such as, arriving at a breakpoint).

**TVDSVR PROCESS:** The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

**UNDIVING:** The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the **undive** icon in the upper right-hand corner of the window.

**UPPER BOUND:** The last element in the dimension of an array or the slice of an array.

**USER INTERRUPT KEY:** A keystroke used to interrupt commands, most commonly defined as **^C** (Ctrl-C).

**VARIABLE WINDOW:** A TotalView window displaying the name, address, data type, and value of a particular variable.

**VISUALIZER PROCESS:** A process that works with TotalView in a separate window, allowing you to see a graphical representation of program array data.

**WATCHPOINT:** An action point specifying that execution should stop whenever the value of a particular variable is updated.

## Citations

**LLNL:** This definition was taken from documentation residing on the web site of the Lawrence Livermore National Laboratories. [www.llnl.gov](http://www.llnl.gov)

:



# Index

## Symbols

\$clid intrinsic 234  
\$count intrinsic 218, 221, 236  
\$countall intrinsic 236  
\$countthread intrinsic 236  
\$debug assembler pseudo op 243  
\$denorm filter 190  
\$duid intrinsic 234  
\$hold assembler pseudo op 243  
\$hold intrinsic 237  
\$holdprocess assembler pseudo op 243  
\$holdprocess intrinsic 237  
\$holdprocessall intrinsic 237  
\$holdprocesstopall assembler pseudo op 243  
\$holdstopall intrinsic 237  
\$holdthread assembler pseudo op 244  
\$holdthread intrinsic 237  
\$holdthreadstop assembler pseudo op 244  
\$holdthreadstop intrinsic 237  
\$holdthreadstopall assembler pseudo op 244  
\$holdthreadstopall intrinsic 237

\$holdthreadstopprocess assembler pseudo op 244  
\$holdthreadstopprocess intrinsic 237  
\$inf filter 189  
\$long\_branch assembler pseudo op 244  
\$nan filter 189  
\$nanq filter 189  
\$nans filter 189  
\$ndenorm filter 190  
\$newval intrinsic 230, 235  
\$nid intrinsic 235  
\$ninf filter 190  
\$oldval intrinsic 230, 235  
\$pdenorm filter 190  
\$pid intrinsic 235  
\$pinf filter 190  
\$processduid intrinsic 235  
\$stop assembler pseudo op 244  
\$stop intrinsic 221, 231, 237  
\$stopall assembler pseudo op 244  
\$stopall intrinsic 237  
\$stopprocess assembler pseudo op 244  
\$stopprocess intrinsic 237

\$stopthread assembler pseudo op 244  
\$stopthread intrinsic 237  
\$systid intrinsic 235  
\$tid intrinsic 235  
\$value intrinsic 192  
\$visualize 112, 238  
    in animations 254  
    in expressions 253  
    using casts 253  
\$visualize intrinsic 252  
%C server launch replacement character 308  
%C server launch replacement characters 66  
%C single process server launch command 66  
%D bulk server launch command 68  
%D pathname replacement character 308  
%D single process server launch command 67  
%H bulk server launch command 68  
%H hostname replacement character 308

%L bulk server launch command 68  
 %L host and port replacement character 308  
 %L single process server launch command 67  
 %N bulk server launch command 69  
 %N line number replacement character 308  
 %P bulk server launch command 68  
 %P password replacement character 308  
 %P single process server launch command 67  
 %R single process server launch command 66  
 %S source file replacement character 308  
 %t1 bulk server launch command 69  
 %t1 file replacement character 309  
 %t2 bulk server launch command 69  
 %t2 file replacement character 309  
 %V bulk server launch command 68  
 %V verbosity setting replacement character 309  
 && operator 192  
 . (period)  
     in suffix of process names 136  
 .pghpfrc file 115  
 .rhosts file 70, 85  
 .stb files 279  
 .stx files 279  
 .Xdefaults file 17, 57, 275  
 /proc file system 322  
 /usr/lib/array/arrayd.conf file 69  
 : (colon), in array type strings 163  
 : as array separator 184  
 <address> data type 166  
 <char> data type 166  
 <character> data type 166

<code> 158  
 <code> data type 166, 169  
 <complex\*16> data type 166  
 <complex\*8> data type 166  
 <complex> data type 166  
 <double precision> data type 167  
 <double> data type 166  
 <extended> data type 167  
 <float> data type 167  
 <int> data type 167  
 <integer\*1> data type 167  
 <integer\*2> data type 167  
 <integer\*4> data type 167  
 <integer\*8> data type 167  
 <integer> data type 167  
 <logical\*1> data type 167  
 <logical\*2> data type 167  
 <logical\*4> data type 167  
 <logical\*8> data type 167  
 <logical> data type 167  
 <long long> data type 167  
 <long> data type 167  
 <real\*16> data type 167  
 <real\*4> data type 167  
 <real\*8> data type 167  
 <real> data type 167  
 <short> data type 167  
 <string> data type 162, 168  
 <void> data type 168  
 > (right angle bracket), indicating nested dives 160

## Numerics

4142 default port 65

## A

-a option to totalview command 35, 51, 290  
 absolute addresses, display assembler as 130  
 acquiring processes 87  
     at startup 77  
 Action Point > At Location 205  
 Action Point > At Location Dialog Box figure 205

Action Point > Properties 123, 205, 207, 209, 213, 217, 249  
     deleting barrier points 215  
 Action Point > Properties dialog box 211  
     figure 214  
 Action Point > Properties Dialog Box figure 206, 210  
 Action Point > Save All 232  
 Action Point > Set Barrier 213  
 Action Point > Suppress All 207  
 Action Point Symbol figure 203  
 action points 8  
     barrier points defined 8  
     breakpoint defined 8  
     common properties 202  
     conditional defined 8  
     definition 201  
     deleting 207  
     disabling 207  
     enabling 207  
     evaluation points defined 8  
     ignoring 207  
     list of 25  
     loading automatically 297  
     machine-level 129  
     saving 232, 299  
     suppressing 207  
     types of 8  
     unsuppressing 207  
     watchpoints defined 8  
 Action Points > Properties 202  
 Action Points > Save All command 232  
 Action Points > Suppress All command 207  
 Action Points page 120  
 Action Points pane 25, 207  
 adaptor\_use option 85  
 Add Directories Dialog Box figure 50  
 adding environment variables 58  
 Address Only (Absolute Addresses) figure 130  
 address range conflicts 223  
 address space, shared 6

- addresses
  - changing 171
  - editing 171
  - of machine instructions 172
  - retracing 276
  - specifying in variable window 156
  - tracking in variable window 154
- AIX
  - compiling on 313
  - linking C++ to dbfork library 319
  - linking to dbfork library 318
  - swap space 326
- align assembler pseudo op 244
- allocated arrays, displaying 169, 170
- Alpha
  - architecture 337
  - floating-point registers 338
  - FPCR register 339
  - general registers 338
- Ambiguous Function dialog box 205
- Ambiguous Function Name
  - Dialog Box figure 206
- ambiguous function names 205
- Ambiguous Line dialog box 97, 143, 204
- Ambiguous Line Dialog Box figure 143, 204
- ambiguous locations 205
- ambiguous names 128
- ambiguous source lines 143
- angle brackets, in windows 160
- animation using \$visualize 254
- architectures 337
  - Alpha 337
  - HP PA-RISC 340
  - Intel-x86 348
  - MIPS 352
  - PowerPC 344
  - SPARC 358
- areas of memory, data type 168
- arguments
  - for totalview command 289
  - for tvdsrv command 304
  - in server launch command 66, 71
  - passing to program 35
  - setting 50
- Arguments page 51
- argv, displaying 169
- Array Data Filter by Range of Values figure 191
- array data filtering 193
  - by comparison 188
  - by range of values 190
  - for IEEE values 189
- Array Data Filtering by Comparison figure 189
- Array Data Filtering for IEEE Values figure 191
- array rank 250
- array services handle (ash) 90
- Array Statistics Window figure 195
- array visualization 253
- arrays
  - \$value special variable 192
  - array data filtering 188
  - bounds 163
  - character 168
  - checksum statistic 194
  - colon separators 184
  - count statistic 194
  - deferred shape 178, 183
  - denormalized count statistic 195
  - display subsection 164
  - displaying 183
  - displaying allocated 170
  - displaying argv 169
  - displaying contents 27
  - displaying declared 169
  - displaying one element 187
  - displaying slices 183
  - diving into 159
  - editing dimension of 164
  - examining data of 9
  - extent 164
  - filter conversion rules 193
  - filter expressions 190
  - filtering 164, 188, 189
  - filtering data 193
  - filtering options 188
  - in C 163
  - in Fortran 163
  - infinity count statistic 195
  - laminating 198
  - limiting display 186
  - lower adjacent statistic 195
  - lower bound 163
  - lower bound of slices 184
  - lower bounds 163
  - maximum statistic 195
  - mean statistic 196
  - median statistic 196
  - minimum statistic 196
  - multidimensional slices 185
  - NaN statistic 196
  - non-default lower bounds 164
  - overlapping nonexistent memory 183
  - pointers to 163
  - quartiles statistic 196
  - reversed indexing of 184
  - skipping elements 185
  - skipping over elements 184
  - slice example 184, 185
  - slices with the variable command 186
  - sorting 188, 193
  - standard deviation statistic 196
  - statistics 194
  - stride elements 184
  - subsections 183
  - sum statistic 196
  - type strings for 163
  - upper adjacent statistic 196
  - upper bound 163
  - upper bound of slices 184
  - visualizing 251
  - visualizing data 11
  - zero count statistic 196
- arrow over line number 25
- Ascending command 193
- ascii assembler pseudo op 244
- asciz assembler pseudo op 244
- ash (array services handle) 90

- ask on dlopen option 331
  - ask\_on\_dlopen option 290, 333
  - ASM icon 202, 208
  - assembler
    - absolute addresses 130
    - and –g compiler option 26
    - constructs 241
    - display symbolically 277
    - displaying 130
    - examining 129
    - expressions 241
    - in code fragment 8, 216
    - symbolic addresses 130
  - Assembler > By Address 130
  - Assembler > Symbolically 130
  - Assembler and Source
    - Interleaved figure 209
  - Assembler command 129
  - Assembler Only (Symbolic
    - Addresses) figure 131
  - assembler operators, TotalView
    - 243
  - assembler-level action points 202
  - at breakpoint state 44
  - At Location command 205
  - Attach subsets command 117
  - attached page 20, 88, 137, 147, 148
  - Attached Page Showing Process
    - and Thread Status figure 44
  - attached process states 44
  - attached thread states 44
  - attaching
    - remote processes, by diving 38
    - selective 117
    - to a task 109
    - to all 119
    - to HP MPI job 84
    - to job 87
    - to MPICH application 80
    - to MPICH job 80
    - to none 119
    - to PE 87
    - to processes 20, 38, 39, 87, 108, 117
    - to PVM task 108
    - to relatives 41
    - to RMS processes 89
    - to SGI MPI job 90
  - attaching to processes 39
  - attaching TotalView to poe 88
  - attaching using File > New
    - Program 40
  - Auto Visualize, in Directory
    - Window 255
  - auto-launch 62
  - autolaunch 61
    - changing 70
    - disabling 62, 63, 70
    - not working 65
    - sequence 71
  - automatic group creation 7
  - automatic process acquisition 7, 77, 80, 84, 107
  - autoRetraceAddresses X resource 276
- ## B
- B state 44
  - background color 276
  - background option 290
  - backgroundColor X resource 276
  - barr\_stop\_all option 290
  - barrier breakpoint 5, 148, 212
    - defined 5, 8, 212
    - states 213
  - barrier breakpoints
    - see also* breakpoints
    - see also* process barrier
    - breakpoint
  - barrier points
    - clearing 207
    - deleting 215
    - stopped process 215
  - base window, defined 159
  - baud rate
    - for serial line 73
    - specifying 306
  - bg option 290
  - bit fields 161
  - bkeepfile option 319
  - Block Distributed Array on Three
    - Processes figure 113
  - blocking send operations 95
  - bounds for arrays 163
  - boxed line number 22, 25, 203
  - branch out instruction 221
  - Breakpoint at Assembler
    - Instruction figure 208
  - breakpoints
    - and MPI\_Init() 87
    - apply to all threads 202
    - automatically copied from
      - master process 80
    - barrier 5
    - barrier defined 5
    - behavior when reached 208
    - changing for parallelization
      - 120
    - clearing 19, 207
    - conditional 216, 218, 236
    - copy, master to slave 80
    - countdown 218, 236
    - counting down 236
    - default stopping action 120
    - defined 8, 201
    - deleting 207
    - disabling 207
    - enabling 207
    - entering 90
    - example setting in
      - multiprocess program 212
    - fork() 211
    - ignoring 207
    - in child process 209
    - in multiple outlines routines
      - 97
    - in parent process 209
    - in spawned process 107
    - listing 25
    - machine-level 129, 208
    - multiple processes 209
    - not shared in separated
      - children 211
    - placing 25
    - popping Process window 280
    - reloading 86
    - removed when detaching 42
    - removing 19

- saving 232
  - set while a process is running 203
  - set while running parallel tasks 86
  - setting 19, 22, 86, 203, 209
  - setting for HPF 115
  - shared by default in processes 211
  - sharing 5, 209, 211
  - stop all related processes 209
  - suppressing 207
  - thread-specific 235
  - toggle location at source-line number 205
  - toggle 205
  - while stepping over 144
  - bss assembler pseudo op 244
  - built-in statements, *see* intrinsics
  - built-in type strings 166
  - bulk launch 308
    - command 63
    - enabling 63
  - Bulk Launch page 65
  - bulk server launch 61, 63
    - on IBM RS/6000 69
    - on SGI MIPS 68
  - bulk server launch command
    - %D 68
    - %H 68
    - %L 68
    - %N 69
    - %P 68
    - %t1 69
    - %t2 69
    - %V 68
    - callback\_host 68
    - callback\_ports 68
    - set\_pws 68
    - verbosity 68
    - working\_directory 68
  - By Address command 130
  - byte assembler pseudo op 244
- C**
- C language
    - array bounds 163
    - arrays 163
    - file suffixes 16
    - filter expression 190
    - how data types are displayed 162
    - in code fragment 8, 216
    - in evaluation points 238
    - type strings supported 162
    - type strings, parameter in .Xdefaults file 277
  - C shell 323
  - C++
    - changing class types 173
    - demangler 293
    - display classes 172
    - in code fragment 8
    - including libdbfork.h 319
    - templates, ambiguous source lines in 204
  - C++ Type Cast to Base Class
    - Dialog Box figure 174
  - C++ Type Cast to Derived Class
    - Dialog Box figure 174
  - call stack 25
  - call tree 11
    - updating display 126
  - Call Tree command 126
    - callback option 303, 304
    - callback\_host 304
    - callback\_host bulk server launch command 68
    - callback\_option single process server launch command 67
    - callback\_ports 304
    - callback\_ports bulk server launch command 68
  - case-sensitivity in searches 281
  - casting 9, 161, 163
    - examples 169
    - to type 158
    - types of variable 161
  - Casting Code figure 159
  - ch\_lfshmem device 78
  - ch\_mpl device 78
  - ch\_p4 device 78, 81, 122
  - ch\_shmem device 78, 81
  - changing
    - autolaunch options 62
    - global variables 139
    - program control groups 138
    - values 28
    - variables 161
  - char data type, retaining data as 168
  - character arrays 168
  - chasing pointers 159
  - checksum array statistic 194
  - child process names 136
  - children calling `execve()`, *see* `execve()`
  - classes, displaying 172
  - Clear All STOP and EVAL command 207
  - clear the continuation signal 149
  - clearing
    - breakpoints 19, 207, 209
    - evaluation points 19
  - CLI 7
    - initialization 17
    - initialization file 17
    - s switch 17
    - starting 16, 35
  - \$clid intrinsic 234
  - Close command 27, 158
  - Close Relatives command 27
  - Close Similar command 27, 158
  - Close, in Data Window 256
  - closed loop, *see* closed loop
  - closing variable windows 158
  - cluster ID 234
  - code constructs supported
    - Assembler 241
    - C 238
    - Fortran 239
  - <code> data type 169, 171
  - code fragments 216, 234
    - modifying instruction path 216
    - when executed 216
    - which programming languages 216
    - within evaluation 8
  - colons as array separators 184
  - color in foreground 278

- comm assembler pseudo op 244
- Command Line 17
- command line arguments 51
  - passing to TotalView 35
- command line option, launch
  - Visualizer 266
- commands 35
  - Action Points > Save All 232
  - Action Points > Suppress All 207
  - arguments 50
  - change Visualizer launch 250
  - Clear All STOP and EVAL 207
  - Create Process (without starting it) 139
  - Detach from Process 41
  - dmpirun 82, 83
  - dpvm 106
  - File > Close 158
  - File > Search Path 248
  - Group > Delete 96
  - Group > Go 138
  - group or process 121
  - input and output files 51
  - mpirun 84, 89, 117
  - New Program 40, 308
  - pgbpf 116
  - poe 79, 85, 114
  - Process > Go 83, 88, 89, 139
  - Process > Startup Parameters 51
  - prun 88
  - pvm 104, 106
  - Quit Debugger 31
  - quitting TotalView 31
  - remsh 70
  - Reset View 263
  - rsh 70, 85
  - server launch, arguments 66
  - Set Search Directory 273, 274, 281
  - Set Signal Handling Mode 105
  - single-stepping 143
  - Thread > Go 139
  - Thread > Set PC 150
  - Tools > Evaluate 233
  - Tools > Message Queue 92
  - Tools > Statistics 194
  - totalview 16, 35, 82, 86, 89
    - command-line options 275
    - core files 35, 42
    - syntax and use 289
  - totalviewcli 16, 35
  - tvdsrv 61
    - launching 66
    - syntax and use 303
  - View > Lookup Variable 155, 177
  - View > Source As > Interleaved 149
  - View > Variable 100
  - visualize 250, 251, 266
  - xrdb 274, 275
- common block
  - displaying 174
  - diving on 174
  - if composite object 175
  - initial address of 175
  - members have function scope 174
  - Multiple tag 175
- Compaq Tru64 UNIX
  - /proc file system 322
  - linking to dbfork library 317
  - swap space 324
- compiled expressions 220, 221
  - allocating patch space for 221
  - performance 220
- compileExpressions X resource 277
- compiler\_vars option 292
- compilers
  - KCC 279
  - mpcc\_r 92
  - mpxlf\_r 92
  - mpxlf90\_r 92
- compilerVars X resource 276
- compiling
  - considerations 34
  - debugging symbols 311
  - g compiler option 15, 34, 311
  - HPF code 116
  - multiprocess programs 33
  - O option 34
  - on Compaq Tru64 UNIX 312
  - on HP-UX 312
  - on IRIX 314
  - on SunOS 315
  - optimization 34
  - options 311
  - programs 15, 33
  - recompiling 38
- compound objects 165
- conditional breakpoints 216, 218, 236
  - defined 8
- conditional watchpoints, *see* watchpoints
- conf file 69
- configure command 79
- configuring for the Visualizer 249
- connecting to a remote host 38
- connection directory 308
- connection for serial line 72
- connection timeout 62, 64
  - altering 62
- console output for tvdsrv 305
- contained functions 177
- context menus 19
- context-sensitive help 13
- continuation signal 148
  - clearing 149
- Continuation Signal command 42, 148
- continuing with a signal 148
- contour lines 263
- contour settings 260
- control groups 135
  - discussion 136
- control registers 151
  - interpreting 151
- conversion rules for filters 193
- Copy command 28
- copying between windows 28
- core dump, naming the signal
  - that caused 43
- core files
  - can only debug local 43
  - examining 7, 42
  - in totalview command 35, 42
  - loading 37

- correcting programs 219
  - count array statistic 194
  - \$count intrinsic 236
  - \$countall intrinsic 236
  - countdown breakpoints 218, 236
  - \$countthread intrinsic 236
  - CPU registers 151
  - cpu\_use option 85
  - Create Checkpoint command 151
  - Create Process (without starting it) command 139
  - creating groups 138
  - creating processes 50, 138
    - and starting them 138
    - errors 270
    - using Step 139
    - without starting them 139
  - crt0.o module 108
  - cTypeStrings X resource 277
  - current data size limit 325
  - current location of program counter 25
  - current queue state 90
  - current stack frame 132
  - current working directory 49, 50
  - customizing TotalView 275
  - Cut command 28
- D**
- d\_process object 322
  - data
    - displaying 9
    - examining 9
    - manipulating 9
    - viewing, from Visualizer 257
  - data assembler pseudo op 244
  - data pane, laminated 198
  - data size limit in C shell 323
  - data types
    - see also* TotalView data types
    - <string> 162
    - C++ 172
    - changing 9, 161
    - changing class types in C++ 173
    - chars, retaining as 168
    - for visualization 250
    - int 162
    - int\* 162
    - int[] 162
    - opaque data 171
    - pointers to arrays 163
    - predefined 166
    - to visualize 250
    - user-defined 174
  - data watchpoints, *see* watchpoints
  - data window 255
    - scaling 260
    - translating 260
    - Visualizer, display commands 256
    - zooming 260
  - data\*pick\_message.background X resource 285
  - dataset
    - deleting 254
    - for Visualizer 250
    - selecting 254
    - showing parameters 263
  - dbfork library 34, 211
    - linking with 34, 317
    - syntax 293
  - dbfork option 292
  - dcheckpoint CLI command 151
  - deadlocks 142
  - deadlocks, message passing 92
  - \$debug assembler pseudo op 243
  - debug, using with MPICH 96
  - debug\_file option 293, 305
  - debugger initialization 17
  - debugger initialization file 17
    - see also* initialization
  - debugger server 61, 303
    - see also*, tvdsrv
    - starting manually 65
  - Debugger Unique ID (DUID) 234
  - debugging
    - distributed programs 10
    - executable file 35
    - HPF code 117, 296
    - multiprocess programs 34
    - not compiled with -g 16
    - OpenMP applications 96
    - programs that call execve 34
    - programs that call fork 34
    - PVM applications 103
    - QSW RMS 88
    - SHMEM library code 110
  - Debugging a Distributed Program with TotalView, figure 3
  - Debugging a Remote Program with TotalView, figure 2
  - debugging Fortran modules 177
  - debugging on a remote host 37
  - debugging over a serial line 72
  - debugging PE applications 84
  - debugging PVM applications 104
  - debugging setuid programs 272
  - declared arrays, displaying 169, 170
  - decw\$sm\_general.dat 275
  - def assembler pseudo op 244
  - default address range conflicts 223
  - default font 278
  - deferred shape array definition 183
  - deferred shape array types 178
  - delay slot instructions for MIPS 357
  - Delete command 28, 38, 121, 150
  - Delete, in Data Window 256
  - deleting
    - action points 207
    - datasets 254
    - processes 217
    - programs 150
  - demangler option 293
  - denorm filter 190
  - denormalized count array statistic 195
  - DENORMs 188
  - Descending command 194
  - Detach command 38, 42
  - Detach from Process command 41
  - detaching from a process 42
  - detaching from processes 41
  - detaching removes all breakpoints 42

- dimmed information, in the root window 147
- Dimmed Process Information in the Root Window figure 148
- directories, setting order of search 48
- directory search path 105
- Directory Window, menu commands 254
- directory\*auto\_visualize.set X resource 285
- Directory, in Data Window 256
- directory.width X resource 285
- disabling
  - action points 207
  - autolaunch 62, 70
  - autolaunch feature 63
  - PVM support 105, 106, 277, 281, 298
- disabling visualization 249
- disassembly, in variable window 172
- discard mode for signals 48
- discarding signal problem 48
- Display of Random Data figure 261
- display option 293
- displayAssemblerSymbolically X resource 277
- displaying 27
  - areas of memory 156
  - argv array 169
  - array data 27
  - arrays 183
  - common blocks 174
  - data 9
  - declared and allocated arrays 169, 170
  - Fortran data types 174
  - Fortran module data 175
  - global variables 155
  - HPF distributed array node 296
  - machine instructions 157, 171
  - memory 156
  - pointer 27
    - into local variables 153
    - into MPI buffer 94
    - into MPI processes 93
    - into parameters 153
    - into processes 40
    - into PVM tasks 108
    - into registers 153
    - into the PC 158
    - into threads 25
    - nested 27
    - nested dive defined 159
    - pointer 27
    - processes 26
    - replacing contents 160
    - routines 26
    - threads 26
    - variable 27
    - variables 27
- displaying a process window 26
- Displaying a Union figure 165
- Displaying C++ Classes that Use Inheritance figure 172
- Displaying Long STL Names figure 157
- displaying long variable names 155
- Dist (distributed) indicator 112
- distributed debugging 10
  - see also* PVM applications
  - remote server 61
- Dive 9
- Dive Anew 27
- dive mouse button 26
- dive stack 160
- Dive Thread command 181
- Dive Thread New command 181
- diving 19, 26, 87, 90
  - from groups page 138
  - in a laminated pane 198
  - in a variable window 159
  - in source code 128
  - into a pointer 27, 159
  - into a process 26
  - into a stack frame 27
  - into a structure 159
  - into a thread 26
  - into a variable 27
  - into an array 159
  - into formal parameters 153
  - into Fortran common blocks 175
  - into function name 128
  - into functions 9
  - into global variables 155
- dlopen 331
- DMPI 92
- dmpirun command 82, 83
- double assembler pseudo op 244
- double-clicking 9
- DPVM
  - see also* PVM
  - enabling support for 106
  - must be running before TotalView 106
  - starting session 106
- dpvm command 106
- dpvm option 106, 294, 305
- dpvm option 106
- DPVMDebugging X resource 277
- drestart CLI command 151
- DUID 234
  - of process 235
- \$duid intrinsic 234
- dump\_core option 294
- Duplicate Base command 27, 160
- Duplicate command 27, 161



- dynamic call tree 11, 126
  - dynamic libraries, debugging in
    - PVM 110
  - dynamic library support
    - limitations 334
  - dynamic option 294
  - dynamic patch space allocation
    - 222
  - dynamically linked, stopping after
    - start() 108
  - dynamically loaded libraries 114, 331
- ## E
- E state 45
  - Edit > Copy 28
  - Edit > Cut 28
  - Edit > Delete 28
  - Edit > Find 28
  - Edit > Find Again 28
  - Edit > Find Dialog Box figure 29
  - Edit > Paste 28
  - edit mode 19
  - Edit Source command 127, 132
  - editing
    - addresses 171
    - laminated pane 198
    - source text 132
    - text 28
    - type strings 161
  - Editing argv figure 170
  - editing compound objects or
    - arrays 165
  - Editing Cursor figure 28
  - editor launch string 132
    - changing 132
  - ELOG icon
    - for event points 19
  - enabling
    - action points 207
    - PVM support 105, 106, 277, 281, 298
  - Environment page 58
  - environment variables 58
    - adding 58
    - adding new ones to
      - environment 58
  - before starting poe 85
  - how to enter 58
  - LD\_LIBRARY\_PATH 317, 318, 320
  - MP\_ADAPTOR\_USE 85
  - MP\_CPU\_USE 85
  - MP\_EUIDEVELOP 94
  - PGI 114
  - TVDSVRLAUNCHCMD 66
  - equiv assembler pseudo op 244
  - error state 44, 45
  - errors 269
    - in multiprocess program 47
  - EVAL (Evaluate Expression)
    - button 217
  - EVAL icon 19
    - for evaluation points 19
  - EVAL point, *see* evaluation points
  - Evaluate command 232, 233, 234, 249
  - evaluating an expression in a
    - watchpoint 225
  - evaluating expressions 232, 233
  - evaluation points 216
    - assembler constructs 241
    - C constructs 238
    - clearing 19
    - commands 236
    - defined 8, 202
    - defining 216
    - examples 218
    - Fortran constructs 239
    - HPF restriction 112
    - listing 25
    - lists of 25
    - machine level 129, 216
    - saving 217
    - setting 19, 217
    - where generated 216
  - event log window 59
  - event points listing 25
  - examining
    - process groups 137
    - source and assembler code
      - 129
    - stack trace and stack frame
      - 153
    - status and control registers
      - 151
  - examining core files 42
  - examining data 9
  - examining processes 135
  - Example of Control Groups and
    - Share Groups figure 137
  - exception data on Compaq Tru64
    - 316
  - exception enable modes 151
  - executables
    - debugging 35
    - reloading 38
  - executing
    - out of function 146
    - to a selected line 144
    - to the completion of a
      - function 146
  - executing a start-up file 17
  - execution context, private 6
  - execution stack, thread private 6
  - execve() 2, 34, 39, 135, 136, 211, 317
    - attaching to processes 39
    - call failed 270
    - debugging programs that call
      - 34
    - failure of 270
    - setting breakpoints with 211
  - Exit command 31
  - Exit, from Visualizer 254
  - exiting TotalView 31
  - expanding path names 138
  - expression evaluation window
    - compiled and interpreted
      - expressions 220
    - discussion 232
  - expression system
    - AIX 335
    - Alpha 335
    - IRIX 335
  - expressions 209
    - can contain loops 232
    - compiled 221
    - evaluating 232
    - performance of 220
    - using 8

–ext option 295  
 extensions, filename 16  
 extent of arrays 164

## F

f77 generated 116  
 fatal errors 323  
 –fg option 295  
 figures  
   Action Point > At Location  
     Dialog Box 205  
   Action Point > Properties  
     Dialog Box 206, 210,  
     214  
   Action Point Symbol 203  
 Add Directories Dialog Box 50  
 Address Only (Absolute  
   Addresses) 130  
 Ambiguous Function Name  
   Dialog Box 206  
 Ambiguous Line Dialog Box  
   143, 204  
 Array Data Filter by Range of  
   Values 191  
 Array Data Filtering by  
   Comparison 189, 191  
 Array Statistics Window 195  
 Assembler and Source  
   Interleaved dialog box  
   209  
 Assembler Only (Symbolic  
   Addresses) 131  
 Attached Page Showing  
   Process and Thread  
   Status 44  
 Block Distributed Array on  
   Three Processes 113  
 Breakpoint at Assembler  
   Instruction Dialog Box  
   208  
 C++ Type Cast to Base Class  
   Dialog Box 174  
 C++ Type Cast to Derived  
   Class Dialog Box 174  
 Casting Code 159  
 Debugging a Distributed  
   Program with TotalView

3  
 Debugging a Remote Program  
   with TotalView 2  
 Dimmed Process Information  
   in the Root Window 148  
 Display of Random Data 261  
 Displaying a Union 165  
 Displaying C++ Classes that  
   Use Inheritance 172  
 Displaying Long STL Names  
   157  
 Diving into Common Block List  
   in Stack Frame Pane  
   175  
 Diving into Local Variables and  
   Registers 154  
 Edit > Find Dialog Box 29  
 Editing argv 170  
 Editing Cursor 28  
 Example of Control Groups  
   and Share Groups 137  
 File > Exit Dialog Box 31  
 File > New Program Dialog  
   Box 36  
 File > Preferences  
   Parallel Page 119  
 File > Preferences Bulk  
   Launch Page 64  
 File > Preferences Dialog Box:  
   Action Points Page 54  
 File > Preferences Dialog Box:  
   Bulk Launch Page 55  
 File > Preferences Dialog Box:  
   Dynamic Libraries Page  
   56  
 File > Preferences Dialog Box:  
   Fonts Page 57  
 File > Preferences Dialog Box:  
   Launch Strings Page 55  
 File > Preferences Dialog Box:  
   Options Page 53  
 File > Preferences Dialog Box:  
   Parallel Page 56  
 File > Preferences Launch  
   Strings Page 249  
 File > Preferences: Server  
   Launch Strings Page 62

File > Properties  
   Action Points Page 210  
 File > Save Pane Dialog Box  
   30  
 File > Signals Dialog Box 47  
 Fortran 90 Pointer Value 180  
 Fortran 90 User Defined Type  
   178  
 Fortran Array with Inverse  
   Order and Limited  
   Extent 186  
 Fortran Modules Window 176  
 Global Variables Window 156  
 Graph Options Dialog Box 259  
 Group > Attach Subset Dialog  
   Box 118  
 Interleaved Source/Assembler  
   (Absolute Addresses)  
   131  
 Laminated Array and Structure  
   199  
 Laminated Scalar Variable 197  
 Manual Launching of  
   Debugger Server 66  
 Message Queue Graph 13  
 Message Queue Graph window  
   91  
 Message Queue Window 93  
 Message Queue Window  
   Showing Pending  
   Receive Operation 94  
 Nested Dive 26  
 Nested Dives 160  
 OpenMP Shared Variable 100  
 OpenMP Stack Parent Token  
   Line 103  
 OpenMP THREADPRIVATE  
   Common Block Variable  
   102  
 Process > Startup Parameters  
   Dialog Box: Arguments  
   Page 51  
 Process > Startup Parameters  
   Dialog Box:  
   Environment Page 59  
 Process > Startup Parameters  
   Dialog Box: Standard

- I/O page 52
- Process and Thread Labels in the Process Window 44
- Process Window 23
- Process Window Tag Field 25
- PVM Tasks and Configuration Window 109
- Resolving Ambiguous Function Names Dialog Box 128
- Root Window: Group Page 137
- Root Window Attached Page 20
- Root Window Groups Page 21
- Root Window Log Page 22
- Root Window Showing Process and Thread Status 72
- Root Window Showing Remote 24
- Root Window Unattached Page 21
- Rotating and Querying 257
- Sample Array Visualization 11
- Sample Call Tree 12
- Sample OpenMP Debugging Session 99
- Sample Visualizer Data Windows 256
- SHMEM Sample Session 111
- Slice Displaying the Four Corners of an Array 186
- Sorted Variable Window 194
- Spelling Corrector Dialog Box 30
- Stop Before Going Parallel Question Dialog Box 118
- Stop Job Question Dialog Box 63
- Stopped Execution of Compiled Expressions 222
- Surface Options Dialog Box 262
- Thread > Continuation Signal Dialog Box 42
- Three Dimensional Array Sliced to Two Dimensions 251
- Three Dimensional Surface Visualizer Data Display 262
- Toolbar 133
- Toolbar Combinations 133
- Tools > Call Tree Dialog Box 126
- Tools > Evaluate Dialog Box 233
- Tools > Watchpoint Dialog Box 227
- TotalView Debugger Server 10
- TotalView Debugging Session Over a Serial Line 73
- TotalView Visualizer Connection 248
- TotalView Visualizer Relationships 248
- TotalView Windows 4
- Two Dimensional Surface Visualizer Data Display 261
- Unattached Page 40, 80
- Using an Expression to Change a Value 161
- Using Assembler 242
- Variable Window 252
- Variable Window for a Global Variable 155
- Variable Window for Area of Memory 158
- Variable Window for small\_array 187
- Variable Window with Machine Instructions 158
- View > Lookup Function Dialog Box 127, 129
- View > Lookup Variable Dialog Box 29
- Visualizer Graph Data Window 259
- Visualizer Windows 255
- Waiting to Complete Message Box 234
- Zooming, Rotating, About an Axis 265
- File > Close 27, 158
- File > Close command 158
- File > Close Relatives 27
- File > Close Similar 27, 158
- File > Edit Source 127, 132
- File > Exit 31
- File > Exit Dialog Box figure 31
- File > New Program 35, 36, 38, 39, 40, 42, 65, 74
- File > New Program Dialog Box figure 36
- File > Preferences 52
  - Action Points page 48, 53, 120
  - Bulk Launch page 54, 63, 65
  - Dynamic Libraries page 54
  - Fonts page 54
  - Launch Strings page 54, 62, 132
  - Options page 47
  - Parallel page 54, 119
- File > Preferences Dialog Box:
  - Action Points Page figure 54
- File > Preferences Dialog Box:
  - Bulk Launch Page figure 55
- File > Preferences Dialog Box:
  - Dynamic Libraries Page figure 56
- File > Preferences Dialog Box:
  - Fonts Page figure 57
- File > Preferences Dialog Box:
  - Launch Strings Page figure 55
- File > Preferences Dialog Box:
  - Options Page figure 53
- File > Preferences Dialog Box:
  - Parallel Page figure 56
- File > Preferences Launch Strings Page figure 249
- File > Preferences: Bulk Launch Page figure 64
- File > Preferences: Parallel Page figure 119
- File > Preferences: Server Launch Strings Page figure 62

- File > Properties: Action Points
    - Page figure 210
  - File > Save Pane 30
  - File > Save Pane Dialog Box
    - figure 30
  - File > Search Path 37, 40, 48, 50, 87
    - search order 48
  - File > Search Path command 48
  - File > Signals 46
  - File > Signals Dialog Box figure 47
  - file for start up 17
  - file option to Visualizer 250, 266
  - files
    - .pghpfrc 115
    - .rhosts 85
    - .stb 279
    - .stx 279
    - .Xdefaults 275
    - hosts.equiv 85
    - libdbfork.h 319
    - license.dat 271
  - fill assembler pseudo op 245
  - filter expression, matching 188
  - filtering
    - array data 188, 193
    - array expressions 190
    - by range of values 190
    - conversion rules 193
    - example 189
    - in sorts 194
    - options 188
  - filters
    - \$denorm 190
    - \$inf 189
    - \$nan 189
    - \$nanq 189
    - \$nans 189
    - \$ninf 190
    - \$pdenorm 190
    - \$pinf 190
  - Find Again command 28
  - Find command 28
  - finding
    - functions 127
    - source code 127, 129
    - source code for functions 127
    - fixed\_font\_size option 58
  - float assembler pseudo op 245
  - floating-point format
    - SPARC 343
  - font 278
  - font X resource 278
  - fonts, in .Xdefaults file 278
  - for HP MPI 83
  - for loop 232
  - foreground (text) color 278
  - foreground option 295
  - foregroundColor X resource 278
  - fork() 2, 34, 135, 211, 317
    - debugging programs that call 34
    - setting breakpoints with 211
  - Fortran
    - array bounds 163
    - arrays 163
    - common blocks 174
    - contained functions 177
    - data types, displaying 174
    - debugging modules 177
    - deferred shape array types 178
    - file suffixes 16
    - filter expression 190
    - identifying version 16
    - in code fragment 8, 216
    - in evaluation points 239
    - module data, displaying 175
    - modules 175, 177
    - pointer types 179
    - type strings supported by
      - TotalView 162
    - user defined types 178
  - Fortran 90 Pointer Value figure 180
  - Fortran 90 User Defined Type
    - figure 178
  - Fortran Array with Inverse Order and Limited Extent figure 186
  - Fortran Modules command 176
  - Fortran Modules Window figure 176
  - frame pointer 146, 147
  - function visualization 126
  - functions
    - finding 127
    - returning from 147
    - searching for 9
- ## G
- g compiler option 15, 26, 34, 116, 274
  - generating a symbol table 34
  - global assembler pseudo op 245
  - global variables
    - changing 139
    - displaying 139
    - diving into 155
  - Global Variables Window figure 156
  - global\_types option 295
  - Globals command 155
  - globalTypenames X resource 278
  - Go 22
  - Go command 82, 86, 88, 89, 121, 134, 138
  - goal location 141
  - GOI
    - process group 141
    - thread group 141
  - goto statements 217
  - Graph Data Window 258
  - Graph Options Dialog Box figure 259
  - Graph visualization menu 254
  - graph window, creating 255
  - graph\*lines.set X resource 285
  - graph\*points.set X resource 285
  - Graph, in Directory Window 255
  - graph.width X resource 285
  - graphs
    - manipulating, in Visualizer 260
    - two dimensional 258
  - gridget 130, 208
  - group
    - process 145
    - thread 145
    - when stopped 141
  - Group > Attach Subset Dialog Box figure 118

Group > Attach Subsets 117  
 Group > Delete 38, 121, 150  
 Group > Delete command 96  
 Group > Go 86, 121, 134, 138  
 Group > Go command 138  
 Group > Halt 121, 143  
 Group > Hold 135  
 Group > Next 121  
 Group > Release 135  
 Group > Reload Symbols 35, 38  
 Group > Restart 150  
 Group > Run To 120  
 Group > Share > Halt 133  
 Group > Step 121  
 Group of Interest (GOI) 141  
 groups 104  
   *see also* processes  
   automatically creating 7  
   creating 138  
   examining 135  
   holding processes 135  
   releasing processes 135  
   running 119  
   single-stepping 6  
   starting 138  
   stopping 119  
 Groups page 20, 137  
 group-width stepping commands 141

## H

h held indicator 134  
 -h localhost option 83  
 half assembler pseudo op 245  
 Halt command 121, 133, 143  
 halt commands 133  
 halting  
   groups 133  
   processes 133  
   threads 133  
 handler routine 45  
 handling signals 45, 46, 105, 106, 281, 299  
 held indicator 134  
 held processes 139  
   defined 213  
 help

  not available on Linux Alpha 13  
 Help command 13  
 hexadecimal address, specifying  
   in variable window 156  
 hi16 assembler operator 243  
 hi32 assembler operator 243  
 hold and release 134  
 \$hold assembler pseudo op 243  
 Hold command 135  
 \$hold intrinsic 237  
 hold state 135  
 Hold Threads command 135  
 holding processes 139  
 holding processes and threads 6  
 holding threads 145  
 \$holdprocess assembler pseudo  
   op 243  
 \$holdprocess intrinsic 237  
 \$holdprocessall intrinsic 237  
 \$holdprocessstopall assembler  
   pseudo op 243  
 \$holdstopall assembler pseudo  
   op 243  
 \$holdstopall intrinsic 237  
 \$holdthread assembler pseudo  
   op 244  
 \$holdthread intrinsic 237  
 \$holdthreadstop assembler  
   pseudo op 244  
 \$holdthreadstop intrinsic 237  
 \$holdthreadstopall assembler  
   pseudo op 244  
 \$holdthreadstopall intrinsic 237  
 \$holdthreadstopprocess  
   assembler pseudo op 244  
 \$holdthreadstopprocess intrinsic  
   237  
 host machine, defined 10  
 host ports 304  
 hostname  
   abbreviated in root window 22  
   expansion 308  
   for tvdsrv 35, 304  
   in square brackets 22  
   replacement 308  
 hosts.equiv file 85

  how TotalView determines share  
   group 138

## HPF

  applications 111  
   compiling for debugging 116  
   debugging 117  
   display node of array element  
     279  
   Dist (distributed) indicator 112  
   enable debugging at source  
     level 278  
   evaluation points restriction  
     112  
   MPI not default 115  
   MPICH 115  
   Rep (replicated) 112  
   search order 114  
   setting breakpoints 115  
   starting programs 116  
   starting TotalView 113  
   starting with MPICH 117  
 -hpf option 279, 296  
 hpf X resource 278  
 -hpf\_node option 296  
 hpfNode X resource 279  
 HP-UX  
   architecture 340  
   shared libraries 331  
   swap space 325  
 hung processes 39

## I

I state 45  
 IBM MPI 84  
 IBM SP machine 78, 79  
 -icc option 296  
 idle state 45  
 Ignore mode warning 48  
 -ignore\_control\_c option 96,  
   272, 296  
 ignoring action points 207  
 indexing, reversed 184  
 indicator 40  
 inet interface name 283  
 inf filter 189  
 infinite loop, *see* loop, infinite  
 infinity count array statistic 195

- INFs 188
- initialization file 17
  - typical contents 17
- initialization search paths 17
- initializing debugging state 17
- initializing the CLI 17
- input files, setting 51
- instructions
  - data type for 169
  - displaying 157, 171
- int data type 162
- int\* data type 162
- int[] data type 162
- Intel-x86
  - architecture 348
  - floating-point registers 350
  - FPCR register 350
    - using 351
  - FPSR register 352
  - general registers 349
- interface name for server 283
- interleave display mode 129
- Interleaved command 129, 149
- interleaved source 208
- Interleaved Source/Assembler (Absolute Addresses)
  - figure 131
- interpreted expressions 220
  - performance 220
- intrinsics 234
  - \$clid 234
  - \$count 218, 221, 236
  - \$countall 236
  - \$countthread 236
  - \$duid 234
  - \$hold 237
  - \$holdprocess 237
  - \$holdprocessall 237
  - \$holdstopall 237
  - \$holdthread 237
  - \$holdthreadstop 237
  - \$holdthreadstopall 237
  - \$holdthreadstopprocess 237
  - \$newval 235
  - \$nid 235
  - \$oldval 235
  - \$pid 235
  - \$processduid 235
  - \$stop 221, 237
  - \$stopall 237
  - \$stopprocess 237
  - \$stopthread 237
  - \$systid 235
  - \$tid 235
  - \$value 192
  - \$visualize 238
    - forcing interpretation 220, 235
- inverting array order 185
- IP over the switch 85
- IRIX
  - /proc file system 322
  - linking to dbfork library 319
  - swap space 327
- J**
  - job\_t::launch 322
- K**
  - K state, unviewable 45
  - kcc\_classes option 296
  - kccClasses X resource 279
  - KeepSendQueue, option 95
  - kernel 45
  - keys, remapping 334
  - keysym 334
  - ksq option 95
- L**
  - labels, for machine instructions 172
  - Laminate > None 197
  - Laminate > Process 197
  - Laminate Threads command. 101
  - Laminated > Thread 197
  - Laminated Array and Structure
    - figure 199
  - Laminated Scalar Variable figure 197
  - laminated variables 197
  - laminating data pane 198
  - lamination
    - arrays and structures 198
    - data panes and Visualizer 252
    - defined 9
    - diving in pane 198
    - editing a pane 198
    - variables 9, 196
  - launch
    - configuring Visualizer 249
    - options for Visualizer 249
    - TotalView Visualizer from
      - command line 266
    - tvdsrv 61, 303
  - Launch Strings page 62, 132
  - lb option 297
  - lcomm assembler pseudo op 245
  - LD\_LIBRARY\_PATH 18
  - left margin area 25
  - left mouse button 18
  - libdbfork.a 317
  - libdbfork.h file 319
  - libraries
    - dbfork 34, 293
    - debugging SHMEM library
      - code 110
    - dynamic 110
    - libtvhpf.so 114
    - loading dynamic 114
    - search order 114
    - shared 294, 329
  - libtvhpf.so library 114
  - license manager problems 272
  - license.dat file 271
  - license.dat file, *see also* TotalView Installation Guide
  - limiting array display 186
  - line most recently selected 147
  - line numbers 25
    - boxed 22
  - linking to dbfork library 317
    - AIX 318
    - C++ and dbfork 319
    - Compaq Tru64 UNIX 317
    - IRIX 319
    - SunOS 5 320
  - Linux swap space 327
  - lists of processes 20
  - LM\_LICENSE\_FILE 18
    - environment variable 271
  - lmgrd 271
  - lo16 assembler operator 243

- lo32 assembler operator 243
- load and loadbind 331
- loading
  - action points 297
  - core file 37
  - file into TotalView 35
  - new executables 36
  - remote executables 37
- local hosts 35
- locations, toggling breakpoints at 205
- Log page 22, 59
- long variable names, displaying 155
- \$long\_branch assembler pseudo op 244
- Lookup Function command 9, 29, 108, 127, 129, 132
- Lookup Variable command 29, 101, 154, 155, 156, 177
  - specifying slides 186
- loop infinite, *see* infinite loop
- lower adjacent array statistic 195
- lower bounds 163
  - non default 164
  - of array slices 184
- lysm TotalView pseudo op 245
- M**
- M state 45
- machine instructions
  - data type 169
  - data type for 169
  - displaying 157, 171
- main() 108
  - stopping before entering 107
- manipulating data 9
- manual hold and release 134
- Manual Launching of Debugger
  - Server figure 66
- master process, recreating slave processes 121
- master thread 97
  - OpenMP 98, 102
  - stack 100
- matching stack frames 197
- maxdsiz\_64 326
- maximum array statistic 195
- maximum data segment size 326
- mean array statistic 196
- median array statistic 196
- memory
  - displaying areas of 156
  - out of, error 272
- memory locations, changing
  - values of 161
- Menu button 19
- menus, context 19
- message passing deadlocks 92
- Message Passing Interface, *see* MPI
- Message Passing
  - Interface/Chameleon
  - Standard, *see* MPICH
- Message Passing Toolkit 92
- Message Queue command 90, 92
- message queue display 89, 95
- Message Queue Graph 12, 90
  - diving 90
  - rearranging shape 92
  - updating 90
- Message Queue Graph command 90
- Message Queue Graph window
  - figure 91
- Message Queue Window
  - figure 93
- Message Queue Window Showing
  - Pending Receive
  - Operation figure 94
- message queues 77
- message states 90
- message tags, reserved 109
- message\_queue option 297
- message-passing programs 120
- messages
  - envelope information 94
  - operations 93
  - reserved tags 109
  - troubleshooting 269
  - unexpected 94
  - verbosity 284
- middle mouse button 19
- minimum array statistic 196
- MIPS
  - architecture 352
  - delay slot instructions 357
  - FCSR register 356
    - using 357
  - floating-point registers 355
  - general registers 353
  - SR register 354
- mixed state 45
- Mkeepftn option 116
- mkswap command 327
- modify watchpoints, *see* watchpoints
- modifying code behavior 216
- module data definition 175
- modules 175, 177
  - debugging, Fortran 177
  - displaying Fortran data 175
- monitoring TotalView sessions 59
- mounting /proc file system 322
- mouse button
  - left 18
  - menu 19
  - middle 19
  - right 19
  - selecting 18
- mouse buttons, using 18
- MP\_ADAPTOR\_USE environment
  - variable 85
- MP\_CPU\_USE environment
  - variable 85
- MP\_EUIDEVELOP environment
  - variable 94
- mpcc\_r compilers 92
- MPI 77
  - acquiring processes at start-up 77
  - attaching to 90
  - attaching to HP job 84
  - attaching to running job 83
  - buffer diving 94
  - communicators 92
  - debugging overview 77
  - library state 92
  - not as default 115
  - on Compaq Alpha 82
  - on HP machines 83

- on IBM 84
- on SGI 89
- process diving 93
- processes, starting 88
- starting on Compaq 82
- starting on SGI 89
- starting processes 82, 89
- troubleshooting 95
- MPI communications library 115
- MPI\_Init() 87, 92
  - breakpoints and timeouts 122
- MPI\_Iprobe() 95
- MPI\_Recv() 95
- MPICH 78, 79
  - and SIGINT 96
  - and the TOTALVIEW
    - environment variable 79
  - attach from TotalView 80
  - attaching to 80
  - ch\_lfshmem device 78, 81
  - ch\_mpl device 78
  - ch\_p4 device 78, 81
  - ch\_shmem device 81
  - ch\_smem device 78
  - configuring 79
  - copy of 78
  - Debugging Tips 122
  - diving into process 80
  - HPF 115
  - MPICH/ch\_p4 122
  - mpirun command 79
  - naming processes 82
  - obtaining 78
  - on workstation clusters 117
  - P4 81
    - p4pg files 81
  - starting TotalView using 79
  - starting using HPF 117
  - tv option 79
  - using -debug 96
- mpirun
  - for HP MPI 84
- mpirun command 79, 84, 89, 117
  - options to TotalView through 122
  - passing options to 122
- mpirun process 90

- MPL\_Init() 87
  - and breakpoints 87
- mpxlf\_r compiler 92
- mpxlf90\_r compiler 92
- mqd option 297
- MQD, *see* message queue display
- Mtotalview option 116, 274
- Mtv option 116
- multiple classes, resolving 128
- multiple outlined routines 97
- multiple sessions 103
- multiple symbol tables 6
- multiprocess programming library 34
- multiprocess programs
  - and signals 47
  - attaching to 41
  - compiling 33
  - process groups 135
  - setting and clearing
    - breakpoints 209
- multithreaded programs 6
- multithreaded signals 148
- mutually recursive functions 147

## N

- n option, of rsh command 71
- n single process server launch
  - command 66
- named groups 20
- names, of processes in process
  - groups 136
- naming MPICH processes 82
- naming rules
  - for control groups 136
  - for share groups 136
- naming the host 304
- NaN array statistic 196
- nan filter 189
- nanq filter 189
- NaNs 188, 189
- nans filter 189
- navigating
  - source code 132
- ndenorm filter 190
- nested dive 26, 27
- Nested Dive figure 26

- nested dive window 160
- nested dive, defined 159
- Nested Dives figure 160
- nested stack frame
  - running to 146
- network debugging 10
- New Base Window
  - in Data Window 256
- New Program command 35, 36, 38, 39, 40, 42, 65, 74, 308
- Next command 121
- Next commands 144
- nicc option 296
- \$nid intrinsic 235
- ninf filter 190
- nlb option 232, 297
- no\_ask\_on\_dlopen option 290, 333
- no\_barr\_stop\_all option 120, 291
- no\_compiler\_vars option 292
- no\_dbfork option 293
- no\_dpvm option 106, 294
- no\_dump\_core option 294
- no\_dynamic option 294, 329
- no\_global\_types option 296
- no\_hpf option 116, 279, 296
- no\_ignore\_control\_c option 296
- no\_kcc\_classes option 296
- no\_message\_queue option 297
- no\_mqd option 297
- no\_parallel option 297
- no\_pop\_at\_breakpoint option 298
- no\_pop\_on\_error option 298
- no\_pvm option 105, 106, 298
- no\_stop\_all option 79, 120, 122, 300
- no\_user\_threads option 300
- node ID 235
- node, attaching from to poe 87
- nodes, HPF 279
- None (laminate) command 197
- None (sort) command 194
- nsb option 299



## O

- O option 34
- offset of window locations 278
- offsets, for machine instructions 172
- \$oldval intrinsic variable 235
- omitting array stride 185
- opaque type definitions 171
- Open process window at
  - breakpoint checkbox 48
- Open process window on signal
  - checkbox 47
- OpenMP 96, 97
  - debugging 97
  - debugging applications 96
  - master thread 97, 98, 101, 102
  - master thread stack context 100
  - on Compaq 98
  - private variables 98
  - runtime library 97
  - shared variables 98, 102
  - stack parent token 102
  - THREADPRIVATE common blocks 101
  - THREADPRIVATE variables 102
  - threads 98
  - viewing shared variables 101
  - worker threads 97
- OpenMP Shared Variable figure 100
- OpenMP Stack Parent Token Line figure 103
- OpenMP THREADPRIVATE
  - Common Block Variable figure 102
- operating systems 321
- operating systems supported 321
- optimizations, compiling for 34
- options
  - for visualize 266
  - ignore\_control\_c 272
  - in Data Window 256
  - Mtotalview 274
  - no\_stop\_all 79
  - patch\_area 223
  - patch\_area\_length 223

- sb 232
- serial 73, 74
- setting 54
- surface data display 263
- tvdsrv
  - callback 304
  - serial 304
  - server 304
  - set\_pw 304
  - user\_threads 300
- org assembler pseudo op 245
- ORNL PVM, *see* PVM
- Out commands 146
- outliers 195, 196
- outlined routine 97, 101, 102
- outlining, defined 97
- output files, setting 51

**P**

- p4 listener process 81
- p4pg files 81
- p4pg option 81
- panes
  - action points list, *see* action points list pane
  - source code, *see* source code pane
  - stack frame, *see* stack frame pane
  - stack trace, *see* stack trace pane
  - width 283
- parallel debugging tips 117
- PARALLEL DO outlined routine 98
- Parallel Environment for AIX, *see* PE
- parallel option 297
- Parallel page 119
- parallel program, restarting 121
- parallel region 97
- parallel tasks, starting 86
- Parallel Virtual Machine, *see* PVM
- passing arguments 35
- passing environment variables to processes 58
- password checking 306

- passwords 306, 307
  - generated by tvdsrv 304
- Paste command 28
- pastings between windows 28
- pastings with middle mouse 19
- patch space
  - static 223
- patch space size, different than 1MB 223
- patch space, allocating 221
- patch\_area\_base option 223, 298
- patch\_area\_length option 223, 298
- patchAreaAddress X resource 280
- patchAreaLength X resource 280
- patching
  - function calls 219
  - programs 218
- PATH environment variable 37, 40, 49
  - for tvdsrv 303
- pathname expansion 138
- pathnames, setting in procgroup file 81
- PC icon 149
- pdenorm filter 190
- PE 84, 87, 92
  - adaptor\_use option 85
  - and slow processes 123
  - applications 84
  - cpu\_use option 85
  - from command line 86
  - from poe 86
  - options to use 85
  - switch-based communication 85
- PE Debugging Tips 122
- pending messages 91
- pending receive operations 93, 94
- pending send operations 93, 95
  - configuring for 95
- pending unexpected messages 93
- performance of interpreted, and compiled expressions 220

- performance of remote
  - debugging 61
- persist option to Visualizer 250, 266
- pghpf command 116
- .pghpfrc file 115
- PGI HPF applications, *see* HPF applications
- \$pid intrinsic 235
- pid.tid to identify thread 24
- pinf filter 190
- pipe for Visualizer 248
- pipng data 30
- platforms 321
- poe 117
  - and mpirun 79
  - and TotalView 86
  - arguments 85
  - attaching to 87, 88
  - command 114
  - interacting with 123
  - on IBM SP 80
  - placing on process list 88
  - required options to 85
  - running PE 86
  - TotalView acquires poe processes 87
- point of execution for
  - multiprocess or multithreaded program 25
- pointer data 27
- pointers 27
  - diving on 27
  - in Fortran 179
  - to arrays 163
  - value of 179
- pop\_at\_breakpoint option 298
- pop\_on\_error option 298
- popAtBreakpoint option 48
- popAtBreakpoint X resource 280
- popping a window 27
- popping process window. 37
- popOnError option 47
- popOnError X resource 281
- port 4142 65, 306
- port number 305
  - for tvdsrv 35, 304
  - replacement 308
  - searching 305
- port option 65, 305
- ports on host 304
- PowerPC
  - architecture 344
  - floating-point registers 346
  - FPSCR register 346
    - using the 348
  - FPSCR register, using 348
  - general registers 344
  - MSR register 345
- predefined data types 166
- Preference
  - Action Points page 48
- Preferences
  - Action Points page 53
  - Bulk Launch page 54, 63, 65
  - Dynamic Libraries page 54
  - Fonts page 54
  - Launch Strings page 54, 62
  - Options page 47
  - Parallel page 54
- preferences
  - overriding 57
  - set using CLI 57
  - setting 54
- Preferences command 52
- preprocessors 295
- primary thread
  - stepping failure 142
- private data for threads 6
- private execution context 6
- private execution stack 6
- private variables 97
  - in OpenMP 98
- proc file system problems 322
- procedures
  - debugging over a serial line 72
  - displaying 170
  - displaying declared and allocated arrays 170
- process
  - detaching 42
  - holding 145
  - releasing 135
  - state 43
  - synchronization 145
- Process > Detach 38, 42
- Process > Go 22, 82, 86, 88, 89, 121, 134, 138, 139
- Process > Go command 83, 88, 89, 139
- Process > Halt 121, 133, 143
- Process > Hold 135
- Process > Hold Threads 135
- Process > Release Threads 135
- Process > Startup Parameters 51, 52
  - Arguments page 51
  - Environment page 58
  - Standard I/O page 52
- Process > Startup Parameters
  - Dialog Box: Arguments Page figure 51
- Process > Startup Parameters
  - Dialog Box: Environment Page figure 59
- Process > Startup Parameters
  - Dialog Box: Standard I/O Page figure 52
- Process and Thread Labels in the Process Window figure 44
- process as dimension in
  - Visualizer 252
- process barrier breakpoint
  - changes when clearing 215
  - changes when setting 215
  - defined 202
  - deleting 215
  - setting 213
- Process command 197
- process DUID 235
- process group 5, 145
  - behavior at goal 145
  - displaying 137
  - when stopped 141
- process ID 235
- process state 25
- process states, attached 44
- process window 5, 22
  - displaying 26
  - host name in title 22

- program counter 25
- raising 47
- updating 38
- Process Window figure 23
- Process Window Tag Field Area figure 25
- process's rank 90
- \$processduid intrinsic 235
- processes
  - see also* automatic process acquisition
  - see also* groups
  - acquiring 80, 81, 107
  - acquiring in PVM applications 104
  - acquisition in poe 87
  - apparently hung 121
  - attaching 20, 38, 39
  - attaching to 39, 87, 108
  - automatic acquisition 7
  - barrier point behavior 215
  - breakpoints shared 209
  - cleanup 110
  - controlling 7
  - copy breakpoints from master process 80
  - creating 50, 138, 139
  - creating by single-stepping 139
  - creating without starting 139
  - definition 7
  - deleting 150
  - deleting related 150
  - detaching from 41
  - dimmed, in the root window 147
  - displaying data 26
  - diving into 40, 87
  - diving on 26
  - error creating 270
  - groups 135
    - changing 138
    - examining 137
  - held 139
  - held defined 213
  - holding 6, 134, 212, 237
  - hung 39
  - killing duplicates 38
  - list of 20
  - loading new executables 36
  - local 39
  - master restart 121
  - MPI 93
  - names 136
  - passing environment variables to 58
  - refreshing process info 134
  - released 213
  - releasing 134, 212, 215
  - remote 39
  - restarting 150
  - single-stepping 6, 140, 141
  - slave, breakpoints in 80
  - starting 22, 139
  - state 43
  - states 44
  - status of 43
  - stop all related 209
  - stopped 213
  - stopped at barrier point 215
  - stopping 133, 216
  - stopping all related 47
  - stopping and deleting 217
  - stopping intrinsic 237
  - stopping spawned 80
  - synchronizing 6
  - types of process groups 135
- procgroup file 81
  - using same absolute path names 81
- program control groups
  - changing 138
  - naming 136
- program counter 25, 40
- program counter (PC) 40
  - arrow icon for PC 25
  - indicator 25
  - setting 149
  - setting program counter 149
  - setting to a stopped thread 149
- program visualization 126
- programs
  - compiling 33
  - compiling using `-g` 15
  - correcting 219
  - deleting 150
  - not compiled with `-g` 16
  - patching 218
  - reloading 35
  - restarting 150
  - setuid, debugging 272
- Properties command 123, 202, 205, 209, 213, 217, 249
- prototypes for temp files 64
- prun command 88
- pthreads, *see* threads
- PVM 305
  - acquiring processes 104
  - attaching procedure 108
  - attaching to tasks 108
  - automatic process acquisition 107
  - cleanup of tvdsrv 110
  - creating symbolic link to tvdsrv 104
  - daemons 109
  - debugging 103
  - debugging dynamic libraries 110
  - disabling support for 105
  - dynamic libraries 110
  - enabling support 277, 281
  - enabling support for 105
  - message tags 109
  - multiple instances not allowed by single user 103
  - multiple sessions 103
  - running with DPVM 104
  - same architecture 108
  - search path 105
  - starting actions 107
  - tasker 107
  - tasker event 107
  - tasks 103, 104
  - TotalView as tasker 103
  - TotalView limitations 103
  - tvdsrv 107
  - Update Command 108
- pvm command 104, 106

PVM groups, unrelated to process groups 104  
 -pvm option 105, 106, 298, 305  
 PVM Tasks and Configuration Window figure 109  
 PVM Tasks command 108  
 pvm\_joining() 110  
 pvm\_spawn() 104, 107  
 pvmDebugging X resource 281  
 pvmgs process 104, 110  
   terminated 110  
 pxd command 331  
 pxd64 command 331

## Q

QSW RMS applications 88  
   attaching to 89  
   debugging 88  
   starting 88  
 quad assembler pseudo op 245  
 Quadrics 88  
 quartiles array statistic 196  
 queue state 90  
 quitting TotalView 31

## R

-r option 299  
 R state 45  
 raising process window 47  
 rank for Visualizer 250  
 ranks 90  
 recompiled executable, reloading 38  
 recompiling 38  
 recursive functions 147  
   single-stepping 146  
 redirecting  
   stdin 52  
   stdout 52  
 registers  
   Alpha FPCR 339  
   editing 151  
   floating-point  
     Alpha 338  
     Intel-x86 350  
     MIPS 355  
     PowerPC 346

  SPARC 360  
   general  
     Alpha 338  
     Intel-x86 349  
     MIPS 353  
     PowerPC 344  
     SPARC 359  
   Intel-x86 FPCR 350  
     using the 351  
   Intel-x86 FPSR 352  
     interpreting 151  
   MIPS FCSR 356  
     using the 357  
   MIPS SR 354  
   Power FPSR 346  
   Power MSR 345  
   PowerPC FPSR 346  
     using 348  
   PowerPC FPSR,  
     using 348  
   PowerPC MSR 345  
   SPARC FPSR 361  
   SPARC FPSR, using 362  
   SPARC PSR 359  
 relatives, attaching to 41  
 Release command 135  
 release state 135  
 Release Threads command 135  
 releasing a process 135  
 Reload Symbols command 35, 38  
 reloading a recompiled executable 38  
 reloading breakpoints 86  
 reloading executables 38  
 reloading programs 35  
 remapping keys 334  
 Remote Debug Server Launch preferences 62  
 remote debugging 61  
   *see also* PVM applications  
   definition 10  
   launching tvdsr 61  
   tvdsr command syntax 303  
 remote executables  
   loading 37  
 remote host  
   connecting to 38

  debugging on 37  
 remote hosts 35  
 remote login 85  
 -remote option 35, 38, 298  
 remote shell command, changing 70  
 removing breakpoints 19  
 remsh command 70, 308  
 remsh command, used in server launches 66  
 Repl (replicated) indicator 112  
 replacement characters 307  
 replacing contents of variable window 160  
 reserved message tags 109  
 Reset command 127, 132  
 Reset View command 263  
 resetting the program counter 149  
 Resolving Ambiguous Function Names Dialog Box figure 128  
 resolving ambiguous names 128  
 resolving multiple classes 128  
 resolving multiple static functions 128  
 resources, for .Xdefaults file 275  
 Restart Checkpoint command 151  
 Restart command 150  
 restarting parallel programs 121  
 restarting programs 150  
 resuming  
   executing thread 149  
   execution 139  
   processes with a signal 148  
 retracing addresses 276  
 returning to original contents 127  
 reusing windows 27  
 reversed indexing 184  
 right angle bracket (>) 26  
 right arrow is program counter 40  
 right mouse button 19  
 RMS applications 88  
   attaching to 89  
   starting 88  
 root window 3, 19, 20

- attached page 20, 88, 137, 147, 148
  - dimmed information 147
  - Groups page 137
  - groups page 20, 137
  - Log page 22, 59
  - selecting a process 26
  - state indicator 43
  - unattached page 20, 38, 39, 43, 45, 80, 87
  - Root Window Attached Page
    - figure 20
  - Root Window Groups Page figure 21
  - Root Window Log Page figure 22
  - Root Window Showing Process and Thread Status figure 72
  - Root Window Showing Remote figure 24
  - Root Window Unattached Page figure 21
  - Root Window: Group Page figure 137
  - Rotating and Querying figure 257
  - rotating surface 263
  - rounding modes 151
  - routine visualization 126
  - routines, diving on 26
  - routines, selecting 25
  - RPM runtime library 113, 116
  - rsh command 70, 85
  - Run To command 120
  - Run To commands 145
  - running groups 119
  - running state 45
  - runtime libraries
    - RPM 113, 116
    - SMP 113, 116
- S**
- S state 45
  - s switch to CLI 17
  - Sample Array Visualization figure 11
  - Sample Call Tree figure 12
  - Sample Message Queue Graph figure 13
  - Sample OpenMP Debugging Session figure 99
  - Sample Visualizer Data Windows figure 256
  - Save All (action points) command 232
  - Save All command 232
  - Save Pane command 30
  - saving
    - action points 232, 299
    - window contents 30
  - sb option 232, 299
  - scaling a surface 264
  - scaling data window 260
  - scrolling 18
    - undoing 132
  - search order, HPF 114
  - Search Path command 37, 40, 48, 50, 87
    - search order 48
  - search paths
    - in .Xdefaults file 281
    - order 48
    - setting 48, 105
  - search paths for initialization 17
  - search\_port option 65, 305
  - searchCaseSensitive X resource 281
  - searching 28
    - for functions 9
    - for source code 129
    - locating closest match 29
    - source code 127
  - Searching, see Edit > Find, View > Lookup Function, View Lookup Variable
  - searching, variable not found 29
  - searchPath X resource 281
  - select button 18
  - selected line, running to 146
  - selecting
    - different stack frame 25
    - routines 25
    - source code, by line 149
    - source line 142
  - selecting text 28
  - sending signals to program 48
  - serial line
    - baud rate 73
    - debugging over a 72
    - radio button 74
    - starting TotalView 74
  - serial line connection 306
  - serial option 73, 74
  - serial option 299, 303, 306
  - server launch 62
    - command 62
    - enabling 62
    - replacement character 66
  - server launch command 308
  - server option
    - not secure 65
  - server option 65, 303, 306
  - servers, number of 308
  - Set Barrier command 213
  - Set PC command 150
  - Set Search Directory command 273, 274, 281
  - Set Signal Handling Mode command 105
  - set\_pw option 304, 306
  - set\_pw single process server launch command 67
  - set\_pws bulk server launch command 68
  - set\_pws option 307
  - setting
    - barrier breakpoint 213
    - breakpoints 19, 86, 203, 209
    - breakpoints while running 203
    - command arguments 51
    - command line arguments 50, 51
    - environment variables 58
    - evaluation points 19, 217
    - HPF defaults 115
    - input and output files 51
    - search path 48
    - search paths 48, 105, 281
    - thread specific breakpoints 235
  - setting options 54

- setting preferences 54
- setting up, debug session 33, 61, 77
- setting X resources 54
- setuid programs 272
- shape arrays, deferred types 178
- Share > Halt 133
- share group 135, 147, 215
  - determining 138
  - determining members of 138
  - discussion 136
  - naming 136
- shared address space 6
- shared libraries 294, 329
  - HP-UX 331
- shared memory library code, *see* SHMEM library code debugging
- shared variables 97
  - in OpenMP 100
  - OpenMP 98, 102
  - procedure for displaying 100
- sharing action points 211
- sharing breakpoints 5
- SHLIB\_PATH 18
- shm option 299
- SHMEM library code debugging 110
- SHMEM Sample Session figure 111
- showing areas of memory 156
- SIGALRM 123
- SIGINT signal 96
- signal
  - clearing 149
- signal handling mode 46
- signal list 48
- signal that caused core dump 43
- signal\_handling\_mode option 299
- signalHandlingMode option 46
- signal\_handling\_mode option 299
- signalHandlingMode X resource 281
- signals
  - affected by hardware registers 46
  - continuing execution with 148
  - default handling behavior 46
  - defining how handled 8
  - discarding 48
  - handler routine 45
  - handling 45
  - handling in PVM applications 105, 106
  - handling in TotalView 45, 281, 299
  - handling mode 46
  - resending 48
  - SIGALRM 123
  - SIGTERM 105, 106
  - stopping 48
- Signals command 46
- SIGSTOP
  - used by TotalView 46
  - when detaching 42
- SIGTERM signal 105, 106
  - stops process 105
  - terminates threads on SGI 98
- SIGTRAP, used by TotalView 46
- single process server launch 61, 62
- single process server launch command
  - %C 66
  - %D 67
  - %L 67
  - %P 67
  - %R 66
  - %verbosity 67
  - callback\_option 67
  - n 66
  - set\_pw 67
  - working\_directory 67
- single-stepping 8, 140, 143
  - commands 143
  - groups 6
  - group-width 141
  - in a nested stack frame 146
  - into function calls 144
  - not allowed for a parallel region 97
  - on primary thread only 140
- operating system
  - dependencies 146, 149
  - over function calls 144
  - process-width 141
  - recursive functions 146
  - slow performance 273
  - threads 142
  - to a selected line 144
- skipping elements 185
- sleeping state 45
- Slice Displaying the Four Corners of an Array figure 186
- slices
  - defining 184
  - descriptions 187
  - displaying one element 187
  - examples 184, 185
  - in sorts 194
  - lower bound 184
  - multidimensional 185
  - of arrays 183
  - operations using 179
  - reversing indexing 184
  - stride elements 184
  - upper bound 184
  - with the variable command 186
- SMP machines 78
- SMP runtime library 113, 116
- sockets 72
- Sort > Ascending 193
- Sort > Descending 194
- Sort > None 194
- Sorted Variable Window figure 194
- sorting
  - array data 193
  - array elements 188
- Source As > Assembler 129
- Source As > Interleaved 129, 149
- Source As > Source 129
- source being interleaved 208
- source code
  - examining 129
  - finding 127, 129
  - navigating 132
- source code pane 25, 273, 283

- Source command 129
- source file suffixes 16
- source lines
  - ambiguous 143
  - editing 132
  - searching 142
  - selecting 142
- source pane 22, 25
- source statements as comments 208
- source-level breakpoints 203
- sourcePaneTabWidth X resource 283
- space allocation
  - dynamic 222
  - static 222, 223
- SPARC
  - architecture 358
  - floating-point format 343
  - floating-point registers 360
  - FPSR register 361
    - using 362
  - general registers 359
  - PSR register 359
- spawned processes, stopping 80
- specifying search directories 50
- spell checker 29
- spellCorrection X resource 283
- spelling checker 283
- Spelling Corrector Dialog Box
  - figure 30
- stack
  - master thread 100
  - trace, examining 153
  - unwinding 150
- stack context of the OpenMP
  - master thread 100
- stack frame
  - current 132
  - examining 153
  - matching 197
  - pane 25
    - selecting different 25
- Stack Frame Pane 158
- stack panes 25
- stack parent token 102
  - diving 102
- stack trace pane 25
  - displaying source 27
- standard deviation array statistic 196
- Standard I/O page 52
- standard input, and launching tvdsrv 71
- start(), stopping within 108
- start\_pes() 110
- starting
  - CLI 16
  - groups 138
  - parallel tasks 86
  - processes 22
  - TotalView 16, 34, 42, 86
  - TotalView for HPF 113
  - tvdsrv 35, 61, 65, 107
- start-up file 17
- Startup Parameters
  - Environment page 58
- Startup Parameters command 51, 52
  - Arguments page 51
  - Standard I/O page 52
- start-up, acquiring processes 77
- state
  - and status 43
  - of processes and threads 43
  - unattached process 45
- state characters 45
- state, initializing 17
- static constructor code 139
- static functions, resolving
  - multiple 128
- static patch space allocation 222, 223
- static patch space assembler
  - code 223
- statically linked, stopping in
  - start() 108
- statistics for arrays 194
- status
  - and state 43
  - of processes 43
  - of threads 43
- status registers
  - examining 151
  - interpreting 151
- stdin, redirect to file 51
- stdout, redirect to file 51
- Step command 121
- Step commands 144
- stepping
  - see also* single-stepping
  - apparently hung 121
  - into 144
  - over 144
  - primary thread can fail 142
  - Run (to selection) Group
    - command 120
  - threads 142
- stepping commands 139
- STL variables, displaying 155
- \$stop assembler pseudo op 244
- Stop Before Going Parallel
  - Question Dialog Box
    - figure 118
- Stop control group on error
  - checkbox 48
- STOP icon 19, 203, 208
  - for breakpoints 19, 203
- \$stop intrinsic 237
- Stop Job Question Dialog Box
  - figure 63
- stop\_all option 299
- \$stopall intrinsic 237
- Stopped Execution of Compiled
  - Expressions figure 222
- stopped process 215
- stopped state 45
  - unattached process 45
- stopping
  - all related processes 47
  - groups 119
  - processes 133, 217
  - spawned processes 80
  - threads 133
- \$stopprocess assembler pseudo
  - op 244
- \$stopprocess intrinsic 237
- \$stopthread intrinsic 237
- stride
  - default value of 185
  - elements 184

- in array slices 184
    - omitting 185
  - string assembler pseudo op 245
  - <string> data type 168
  - string syntax 277
  - strings, searching for by case 281
  - structs
    - see also* structures
    - defined using typedefs 165
    - how displayed 164
  - structures 164
    - see also* structs
    - editing types 162
    - laminating 198
  - subroutines, displaying 26
  - suffixes
    - of processes in process groups 136
    - of source files 16
  - sum array statistic 196
  - SunOS 5
    - /proc file system 322
    - key remapping 334
    - linking to dbfork library 320
    - swap space 328
  - supported platforms 321
  - Suppress All command 207
  - suppressing action points 207
  - surface
    - display 263
    - in directory window 255
    - rotating 263
    - scaling 264
    - translating 264
    - zooming 264
  - Surface Data Window 260
    - display 262
  - Surface Options Dialog Box figure 262
  - Surface visualization window 254
  - surface window, creating 255
  - surface\*auto\_reduce.set X
    - resource 286
  - surface\*contour.set X resource 286
  - surface\*mesh.set X resource 286
  - surface\*shade.set X resource 286
  - surface\*xrt3dViewNormalized X
    - resource 286
  - surface\*xrt3dXMeshFilter X
    - resource 287
  - surface\*xrt3dYMeshFilter X
    - resource 287
  - surface\*xrt3dZoneMethod X
    - resource 286
  - surface\*zone.set X resource 287
  - surface.height X resource 286
  - surface.width X resource 286
  - suspended windows 234
  - swap command 328
  - swap space 272, 323, 328
    - AIX 326
    - Compaq Tru64 324
    - HP-UX 325
    - IRIX 327
    - Linux 327
    - SunOS 328
  - swapon command 327
  - switch-based communication
    - for PE 85
  - switch-based communications 85
  - symbol table 6
  - symbol table debugging
    - information 15
  - symbolic addresses, displaying
    - assembler as 130
  - Symbolically command 130
  - synchronizing processes 6, 145
  - systems supported 321
  - systid 24
  - \$systid intrinsic 235
- T**
- T state 45
  - tab character 283
  - tag field 203, 208
  - tag field area 19, 25
  - target machine, defined 10
  - tasker event 107
  - tasks
    - attaching to 108
    - diving into 108
    - PVM 103
    - starting 86
  - TCP/IP sockets 72
  - temp file prototypes 64
  - templates, ambiguous lines 204
  - testing when a value changes 225
  - text
    - editing 28
    - locating closest match 29
    - saving window contents 30
    - selecting 28
  - text assembler pseudo op 245
  - third party visualizer 248
  - Thread > Continuation Signal 42, 148
  - Thread > Continuation Signal Dialog Box figure 42
  - Thread > Go command 139
  - Thread > Hold 135
  - Thread > Set PC 150
  - thread as dimension in Visualizer 252
  - Thread command 197
  - thread group 145
    - behavior at goal 145
    - when stopped 141
  - thread ID 24
    - system 235
    - TotalView 235
  - thread local storage 101
    - variables stored in different locations 101
  - thread objects
    - displaying 181
  - Thread Objects command 181
  - thread of interest 133
  - Thread of Interest (TOI) 141
  - thread pane 24
  - THREADPRIVATE common block
    - procedure for viewing
    - variables in 101
  - THREADPRIVATE variables 102
  - threads 6
    - controlling 7
    - definition 7
    - differing operating system definition 6
    - dimmed, in the root window 147



- displaying source 26
- diving 25
- diving on 26
- finding window for 25
- holding 6, 145
- ID format 24
- listing 24
- opening window for 25
- private data 6
- releasing 212
- resuming executing 149
- setting breakpoints in 235
- single-stepping 140, 142
- stack trace 25
- state 43
- states 44
- status of 43
- stopping 133
- synchronizing 6
- systid 24
- tid 24
- TotalView's model 7
- user-level 283
- Threads > Set PC command 150
- thread-specific breakpoints 235
- thread-width stepping command 142
- Three Dimensional Array Sliced to Two Dimensions figure 251
- Three Dimensional Surface Visualizer Data Display figure 262
- tid 24
- \$tid intrinsic 235
- timeouts
  - avoid unwanted 122
  - during initialization 87
  - for connection 62
  - TotalView setting 85
- Toggle Node Display 279
- TOI 133
- Toolbar combinations figure 133
- Toolbar figure 133
- toolbar, using 133
- Tools > Call Tree 126
- Tools > Call Tree Dialog Box figure 126
- Tools > Command Line 17
- Tools > Create Checkpoint 151
- Tools > Evaluate 232, 233, 234, 249
- Tools > Evaluate command 233
- Tools > Evaluate Dialog Box figure 233
- Tools > Fortran Modules 176
- Tools > Globals 155
- Tools > Message Queue 90, 92
- Tools > Message Queue command 92
- Tools > Message Queue Graph 90
- Tools > PVM Tasks 108
- Tools > Restart Checkpoint 151
- Tools > Statistics command 194
- Tools > Thread Objects 181
- Tools > Visualize 200
- Tools > Watchpoint 226, 230
- Tools > Watchpoint Dialog Box figure 227
- TotalView
  - and MPICH 79
  - as PVM tasker 103
  - core files 35
  - host machine definition 10
  - HPF default settings 115
  - interactions with Visualizer 247
  - quitting 31
  - starting 16, 34, 35, 42, 86
  - starting on remote hosts 35
  - target machine definition 10
  - thread model 7
  - Visualizer configuration 249
  - visualizing array data 11
- TotalView Assembler Language 241
- TotalView assembler operators
  - hi16 243
  - hi32 243
  - lo16 243
  - lo32 243
- TotalView assembler pseudo ops
  - \$debug 243
  - \$hold 243
  - \$holdprocess 243
  - \$holdprocessstopall 243
  - \$holdstopall 243
  - \$holdthread 244
  - \$holdthreadstop 244
  - \$holdthreadstopall 244
  - \$holdthreadstopprocess 244
  - \$long\_branch 244
  - \$stop 244
  - \$stopall 244
  - \$stopprocess 244
  - \$stopthread 244
  - align 244
  - ascii 244
  - asciz 244
  - bss 244
  - byte 244
  - comm 244
  - data 244
  - def 244
  - double 244
  - equiv 244
  - fill 245
  - float 245
  - global 245
  - half 245
  - lcomm 245
  - lysm 245
  - org 245
  - quad 245
  - string 245
  - text 245
  - word 245
  - zero 245
- totalview command 16, 17, 35, 42, 82, 86, 89, 289
  - a option 51
  - command-line options 275
  - description 289
  - for HP MPI 83
  - options 290
  - starting on a serial line 74
  - synopsis 289
- TotalView data types
  - <address> 166
  - <char> 166

- <character> 166
- <code> 166, 169
- <complex\*16> 166
- <complex\*8> 166
- <complex> 166
- <double precision> 167
- <double> 166
- <extended> 167
- <float> 167
- <int> 167
- <integer\*1> 167
- <integer\*2> 167
- <integer\*4> 167
- <integer\*8> 167
- <integer> 167
- <logical\*1> 167
- <logical\*2> 167
- <logical\*4> 167
- <logical\*8> 167
- <logical> 167
- <long long> 167
- <long> 167
- <real\* 16> 167
- <real\* 4> 167
- <real\* 8> 167
- <real> 167
- <short> 167
- <string> 168
- <void> 168
- TotalView Debugger Server 37, 73
- TotalView Debugger Server, figure 10
- TotalView Debugger Server, *see* tvdsrv
- TotalView Debugging Session
  - Over a Serial Line figure 73
- TOTALVIEW environment variable 79
- TotalView platforms 321
- TotalView program
  - quitting 31
- TotalView Visualizer 254–267
- TotalView Visualizer Connection
  - figure 248
- TotalView Visualizer Relationships
  - figure 248
- TotalView Visualizer
  - see* Visualizer
- TotalView Windows 4
- TotalView windows
  - action point List pane 25
  - editing cursor 28
  - process 22
  - program counter arrow 25
- totalview\*autoRetraceAddresses
  - X resource 276
- totalview\*backgroundColor X
  - resource 276
- totalview\*compileExpressions X
  - resource 277
- totalview\*compilerVars X
  - resource 276
- totalview\*cTypeStrings X
  - resource 277
- totalview\*displayAssembler
  - Symbolically X resource 277
- totalview\*DPVMDDebugging X
  - resource 277
- totalview\*font X resource 278
- totalview\*foregroundColor X
  - resource 278
- totalview\*globalTypenames X
  - resource 278
- totalview\*hpf X resource 116, 278
- totalview\*hpfNode X resource 279
- totalview\*kccClasses X resource 279
- totalview\*overrideRedirect X
  - resource 280
- totalview\*patchAreaAddress X
  - resource 280
- totalview\*patchAreaLength X
  - resource 280
- totalview\*popAtBreakpoint X
  - resource 280
- totalview\*popOnError X resource 281
- totalview\*pvmDebugging X
  - resource 281
- totalview\*searchCaseSensitive X
  - resource 281
- totalview\*searchPath X resource 281
- totalview\*signalHandlingMode X
  - resource 281
- totalview\*sourcePaneTabWidth X
  - resource 283
- totalview\*spellCorrection X
  - resource 283
- totalview\*useInterface X resource 283
- totalview\*userThreads X resource 283
- totalview\*useTransientFor X
  - resource 284
- totalview\*verbosity X resource 284
- totalviewcli command 16, 17, 35
- transient-for windows 284
- translating a surface 264
- translating data window 260
- TRAP\_FPE environment variable
  - on SGI 46
- troubleshooting xvi, 269
  - checkout failed 271
  - error creating new process 270
  - error launching process 270
  - error while deleting target 270
  - HPF source code does not appear 274
  - MPI 95
  - out of memory 272
  - single-stepping is slow 273
  - source code doesn't appear 273
  - tvdsrv fails to appear 274
  - X resources are not recognized 274
- tv option 79
- TVD.breakpoints file 232
- TVDB\_patch\_base\_address
  - object 223
- tvdb\_patch\_space.s 224
- tvdsrv 35, 37, 58, 61, 62, 63, 72, 220, 304, 307
  - attaching to 108

- cleanup by PVM 110
  - editing command line for poe 87
  - fail in MPI environment 96
  - fails to appear 274
  - launching 66
  - launching, arguments 71
  - PATH environment variable 303
  - starting 65
  - starting for serial line 73
  - starting manually 65
  - symbolic link from PVM directory 104
  - with PVM 107
  - tvdsrv command 303
    - description 304
    - options 304
    - password 304
    - starting 61
    - synopsis 303
    - timeout while launching 63, 64
    - use with DPVM applications 305
    - use with PVM applications 104, 305
  - tvdsrv.conf 306
  - TVDSVRLAUNCHCMD
    - environment variable 66, 308
  - Two Dimensional Surface
    - Visualizer Data Display figure 261
  - two-dimensional graphs 258
  - type casting 161
    - examples 169
  - type names 278
  - type strings
    - built-in 166
    - editing 161
    - for opaque types 171
    - parameter in .Xdefaults file 277
    - supported for Fortran 162
  - type, user defined type 178
  - typedefs
    - defining structs 165
    - how displayed 164
    - types supported for C language 162
- ## U
- UDT 178
  - UDWP, *see* watchpoints
  - UID, UNIX 65
  - unattached page 20, 38, 39, 43, 45, 80, 87
  - Unattached Page figure 40, 80
  - unattached process states 45
    - summary 45
  - undive icon 127
  - undiving, from windows 160
  - unexpected messages 91, 94
  - unions 164
    - how displayed 165
  - unsuppressing action points 207
  - unwinding the stack 150
  - Update command 88, 134, 147
  - updating visualization displays 252
  - upper adjacent array statistic 196
  - upper bounds 163
    - of array slices 184
  - USEd information 177
  - useInterface X resource 283
  - user defined data type 174, 178
  - user\_threads option 300
  - userThreads X resource 283
  - user-visible communicators 92
  - useTransientFor X resource 284
  - Using an Expression to Change a Value figure 161
  - Using Assembler figure 242
  - using expressions 8
- ## V
- value field 233
  - values, changing 28
  - variable
    - diving 27
  - variable window 5
    - closing 158
    - displaying 153
    - duplicating 161
    - in recursion, manually refocus 154
    - laminated display 196
    - replacing contents 160
    - stale in pane header 154
    - tracking addresses 154
    - updates to 154
  - Variable Window figure 252
  - Variable Window for a Global Variable figure 155
  - Variable Window for Area of Memory figure 158
  - Variable Window for small\_array figure 187
  - Variable Window with Machine Instructions figure 158
  - variables
    - at different addresses 198
    - changing the value 161
    - changing values of 9, 161
    - displaying all globals 155
    - displaying contents 27
    - displaying long names 155
    - diving 27
    - in modules 176
    - intrinsic, *see* intrinsic variables
    - laminated display 196
    - laminating 9
    - stored in different locations 101
  - verbosity bulk server launch command 68
  - verbosity level 90
  - verbosity option 300, 307
  - verbosity setting replacement character 309
  - verbosity single process server launch command 67
  - verbosity X resource 284
  - View > Assembler > By Address 130
  - View > Assembler > Symbolically 130
  - View > Dive 9
  - View > Dive Anew 27
  - View > Dive Thread 181
  - View > Dive Thread New 181

- View > Laminate > None 197
- View > Laminate > Process 197
- View > Laminate Threads 101
- View > Laminated > Thread 197
- View > Lookup Function 9, 29, 108, 127, 129, 132
- View > Lookup Function Dialog Box figure 127, 129
- View > Lookup Variable 29, 101, 154, 155, 156
  - specifying slices 186
- View > Lookup Variable command 155, 177
- View > Lookup Variable Dialog Box figure 29
- View > Reset 127, 132
- View > Sort > Ascending 193
- View > Sort > Descending 194
- View > Sort > None 194
- View > Source As > Assembler 129
- View > Source As > Interleaved 129, 149
- View > Source As > Interleaved command 149
- View > Source As > Source 129
- View > Variable command 100
- visualization 11
  - deleting a dataset 254
  - display data 247
  - extract data 247
  - translating a surface 264
  - zooming a surface 264
- \$visualize 238, 252–254
- Visualize command 200, 250, 251, 266
- visualize command 266
- \$visualize EVAL 112
- visualize intrinsic 252
- Visualize\* data\*pick\_message.
  - background X resource 285
- Visualize\*directory\*
  - auto\_visualize. set X resource 285
- Visualize\*directory.width X
  - resource 285
- Visualize\*graph\*lines.set X
  - resource 285
- Visualize\*graph\*points.set X
  - resource 285
- Visualize\*graph.width X resource 285
- Visualize\*surface\*auto\_reduce.
  - set X resource 286
- Visualize\*surface\*contour.set X
  - resource 286
- Visualize\*surface\*mesh.set X
  - resource 286
- Visualize\*surface\*shade.set X
  - resource 286
- Visualize\*surface\*xrt3dView
  - Normalized X resource 286
- Visualize\*surface\*xrt3dXMesh
  - Filter X resource 287
- Visualize\*surface\*xrt3dYMesh
  - Filter X resource 287
- Visualize\*surface\*xrt3dZone
  - Method X resource 286
- Visualize\*surface\*zone.set X
  - resource 287
- Visualize\*surface.height X
  - resource 286
- Visualize\*surface.width X
  - resource 286
- Visualizer 11, 199
  - auto launch options, changing 250
  - choosing method for displaying data 257
  - configuring 249
  - configuring launch 249
  - creating graph window 255
  - creating surface window 255
  - data sets to visualize 251
  - data types 250
  - data window 254, 255
  - data window manipulation commands 260
  - dataset defined 250
  - dataset numeric identifier 250
  - dataset parameters 263
  - deleting datasets 254
  - dimensions 252
  - directory window 254
  - disabling 249
  - display not automatically updated 252
  - exiting from 254
  - file option 250, 266
  - graphs, display 258
  - graphs, manipulating 260
  - how implemented 247
  - interactions with TotalView 247
  - laminated data panes 252
  - launch command, changing shell 250
  - launch from command line 266
  - launch options 249
  - method 257
  - method automatically chosen 258
  - new or existing dataset 250
  - number of arrays 251
  - persist option 250, 266
  - pipe 248
  - rank 250
  - relationship to TotalView 248
  - rotating 263
  - scaling a surface 264
  - selecting datasets 254
  - shell launch command 250
  - slices 251
  - surface data display options 263
  - Surface Data Window 260
  - third party 248
  - using casts 253
  - windows, types of 254
- visualizer
  - closing connection to 250
  - customized command for 249
- Visualizer Graph Data Window figure 259
- Visualizer Windows figure 255
- visualizing
  - data 247
  - data sets from a file 266

- from variable window 251
- in expressions using \$visualize 252
- visualizing data 254
- <void> data type 168

## W

- W state 45
- Waiting to Complete Message
  - Box figure 234
- watching memory 228
- Watchpoint command 226, 230
- Watchpoint Properties dialog box 227
- watchpoint state 45
- watchpoints 224
  - \$newval 230
  - \$oldval 230
  - alignment 231
  - conditional 225, 230
  - copying data 229
  - creating 226
  - defined 8, 202
  - disabling 227
  - diving into 227
  - enabling 227
  - evaluated, not compiled 231
  - evaluating an expression 225
  - example of triggering when
    - value goes negative 230, 231
  - length compared to \$oldval or \$newval 231
  - lists of 25

- lowest address triggered 229
- modifying a memory location 225
- monitoring adjacent locations 229
- multiple 229
- not saved 232
- PC position 228
- problem with stack variables 228
- supported platforms 225
- testing a threshold 225
- testing when a value changes 225
- triggering 225, 228
  - watching memory 228
- Window > Duplicate 27, 161
- Window > Duplicate Base 27, 160
- Window > Update 88, 134, 147
- window contents, saving 30
- window location 276
  - offset 278
- windows 158
  - closing 158
  - copying between 28
  - data 255
  - Data Window 256
  - Directory Window 254
  - event log 59
  - graph data 258
  - pasting between 28
  - popping 27
  - problems with 274
  - Surface Data Window 260

- suspended 234
- transient-for 284
- Windows > Update (PVM) 108
- word assembler pseudo op 245
- worker threads 97
- workers group 145
- working directory 50
- working\_directory bulk server
  - launch command 68
- working\_directory option 307
- working\_directory single process
  - server launch command 67

## X

- X resource option 290
- X resources setting 54
- Xdefaults 275
- xrdb command 274, 275
- Xresource=value option 290
- xterm
  - launching tvdsrv from 71
  - problems with 272

## Z

- Z state 45
- zero assembler pseudo op 245
- zero count array statistic 196
- zombie state 45
- zone maps 260
- zooming a surface 264
- zooming data window 260
- Zooming, Rotating, About an Axis
  - figure 265

